

Integers II

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael

Chendur Jayavelu

Nahush Shrivatsa

Renee Ruan

Sean Siddens

Connie Chen

Joshua Tan

Naama Amiel

Rubee Zhao

Waleed Yagoub

Chloe Fong

Nikolas McNamee

Neela Kausik

Samantha Dreussi



<http://xkcd.com/571/>

Relevant Course Information

- ❖ HW3 due Tonight, Friday (10/04) @ 11:59 pm
- ❖ HW4 due Monday (10/07) @ 11:59 pm
- ❖ Lab 1a due Tuesday (10/08) @ 11:59pm
 - Use `pctest` and `d1c.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, due Monday (10/14) @ 11:59pm
 - Bit manipulation on a custom number representation
 - 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:

- U_{\min} , U_{\max} , T_{\min} , T_{\max}
- 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



Review Questions

- $2^6 = 64$ patterns
- $-32 \quad -1 \quad 0 \quad 31$ This complement signed
- Signed
- ❖ What is the value (and encoding) of TMin for a fictional 6-bit wide integer data type?
- $0b \frac{1}{-2^5} \frac{0}{2^4} \frac{0}{2^3} \frac{0}{2^2} \frac{0}{2^1} \frac{0}{2^0} = -32$
- ❖ For unsigned char uc = 0xA1;, what are the produced data for the cast (unsigned short)uc?

unsigned \rightarrow zero extension 0x 00A1

- ❖ What is the result of the following expressions?

- (signed char)uc >> 2
- (unsigned char)uc >> 3

$0b \underline{10} 10 000X$ $\xrightarrow{\text{signed arithmetic}} 0b 1110 1000 = 0xE8$

$0b \sim 10 10 000X$ $\xrightarrow{\text{unsigned logical}} 0b 0001 0100 = 0x14$

Why Does Two's Complement Work?

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

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

- 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:

$$\text{additive inverse} \left\{ \begin{array}{l} \text{bit representation of } x \\ + \text{ bit representation of } -x \end{array} \right\} = 0 \quad (\text{ignoring the carry-out bit})$$

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

$\begin{array}{r} \quad \quad \quad \quad \quad \quad \quad \\ 00000001 \\ - + 11111111 \\ \hline 1 \boxed{00000000} \end{array}$	$\begin{array}{r} 00000010 \quad 2 \\ + 11111110 \quad -2 \\ \hline 10000000 \end{array}$	$\begin{array}{r} 11000011 \quad 61 \\ + 00111101 \quad -61 \\ \hline 10000000 \end{array}$
-----------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------

These are the bitwise complement plus 1!

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

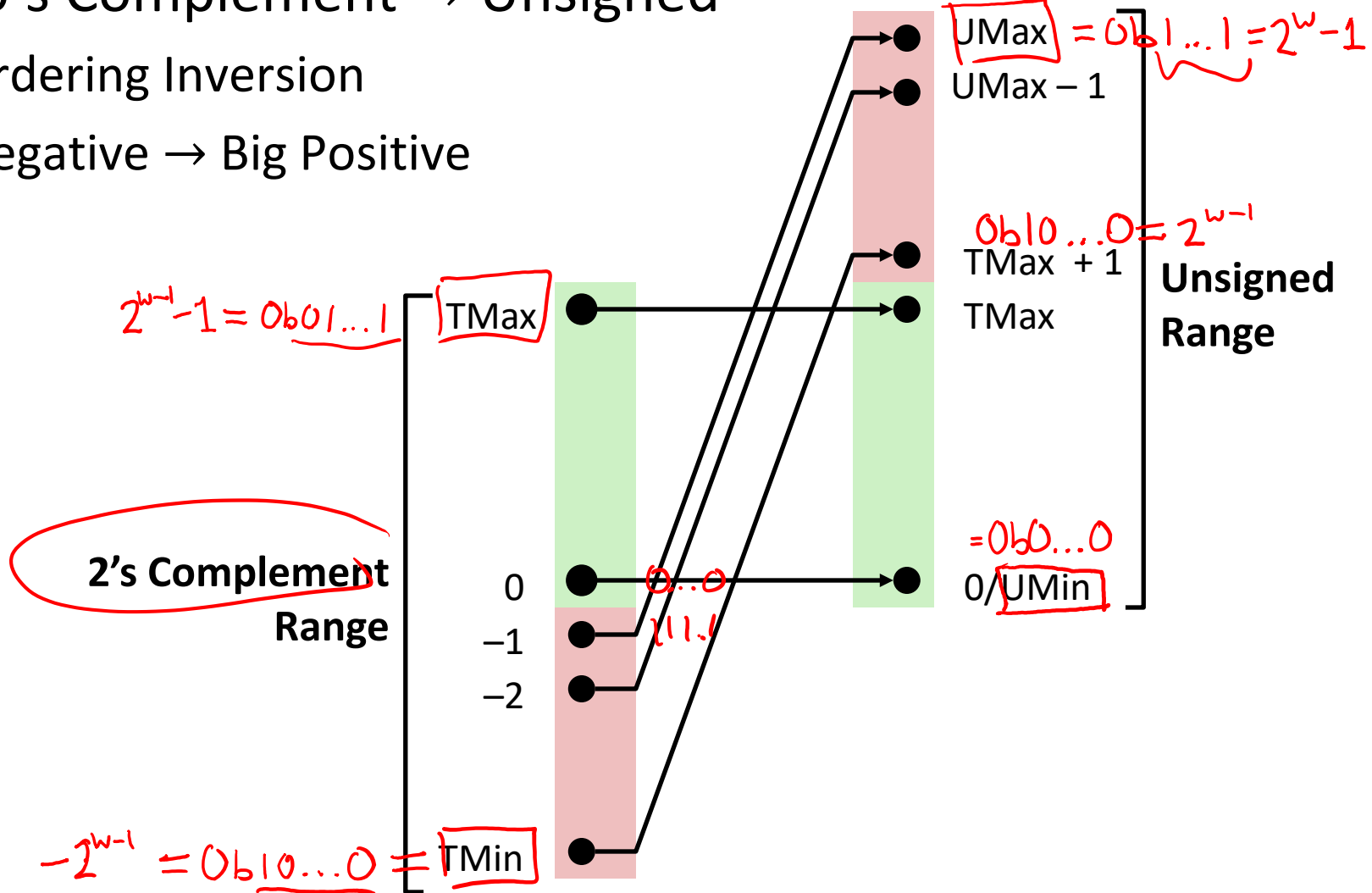
Integers

- ❖ **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 \rightarrow Unsigned

- Ordering Inversion
- Negative \rightarrow 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

Signet →

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;

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

- tx = ux;
- uy = ty;

(also implicitly occurs with printf format specifiers)



7

7

- 7

7

- 7

7

7

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

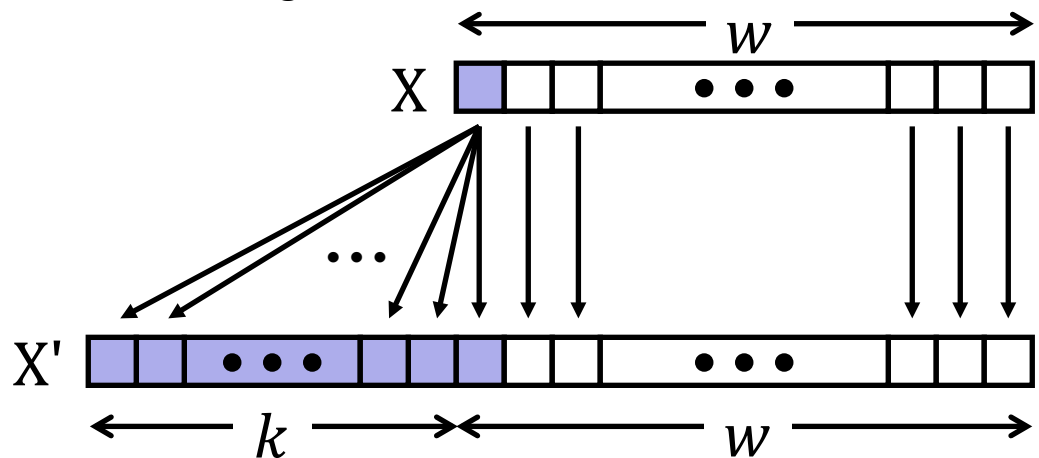
■ Let x_i be the i -th digit of X in binary

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

Signed
4 bit example

$0b1000 = -8$

$0b \overset{-16}{1} \overset{-8}{0} \overset{-4}{0} \overset{-2}{0} \overset{-1}{0} = -8$
 $-16 + 8 = -8$



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

0b 1 1 1 1	<u>Signed</u>	<u>Unsigned</u>
	-1	15
+ 0 0 0 1	+1	+1
<hr/>	<hr/>	<hr/>
1 0 0 0 0	0	16
		↑ overflow

A red arrow points from the '1' in the '1 0 0 0 0' result of the signed addition to the '16' in the unsigned addition, indicating the overflow.

Arithmetic Overflow (Review)

Bits	Unsigned	Signed
0000	0 <i>U_{Min}</i>	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7 <i>T_{Max}</i>
1000	8	-8 <i>T_{Min}</i>
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15 <i>U_{Max}</i>	-1

❖ When a calculation produces a result that can't be represented in the current encoding scheme

- Integer range limited by fixed width *U_{Min} - U_{Max}*
T_{Min} - T_{Max}
- 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!

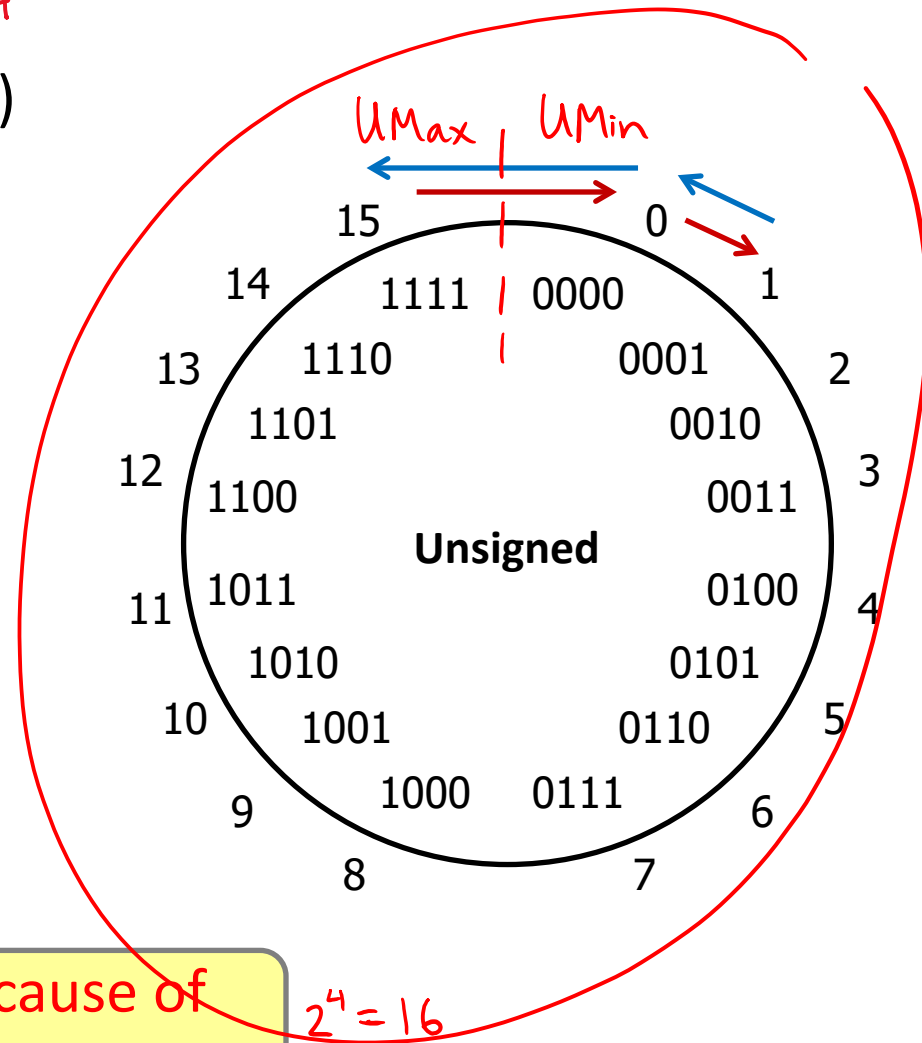
Overflow: Unsigned

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

$$\begin{array}{r}
 \rightarrow 15 \\
 + 2 \\
 \hline
 \textcircled{\text{17}} \\
 1
 \end{array}
 \qquad
 \begin{array}{r}
 1111 \\
 + 0010 \\
 \hline
 \textcircled{10001} \\
 \uparrow
 \end{array}$$

- ❖ **Subtraction:** borrow ($+2^N$)

$$\begin{array}{r}
 1 \\
 - 2 \\
 \hline
 \textcircled{-1} \\
 15
 \end{array}
 \qquad
 \begin{array}{r}
 10001 \\
 - 0010 \\
 \hline
 \textcircled{1111}
 \end{array}$$



±2^N because of
modular arithmetic

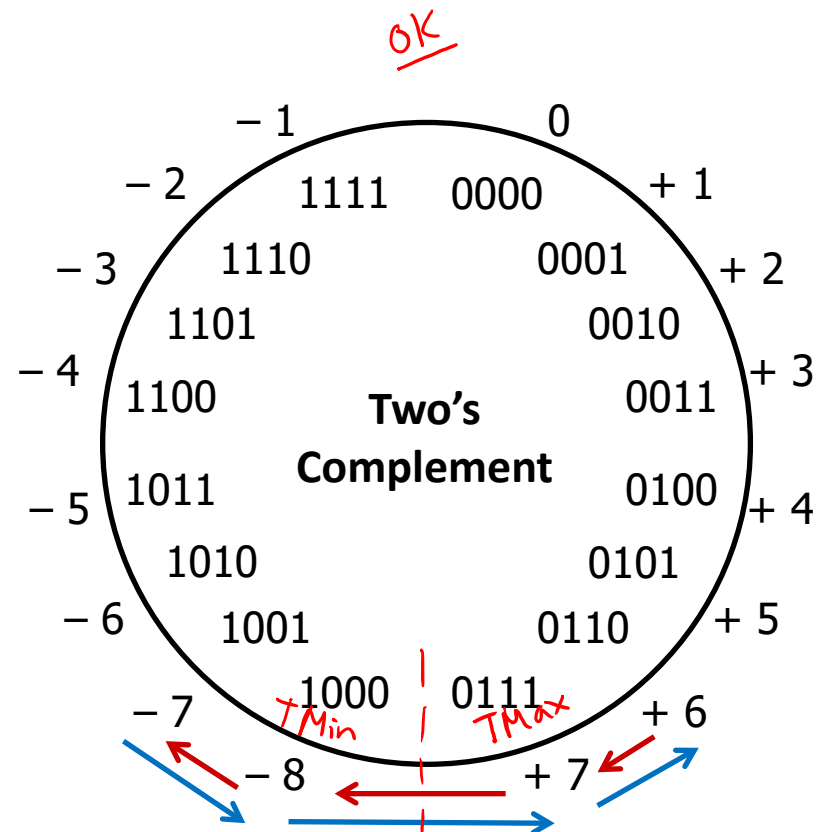
Overflow: Two's Complement

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

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

- ❖ **Subtraction:** (-) + (-) = (+)?

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



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

Practice Questions

❖ Assuming 8-bit integers:

- $0x27 = 39$ (signed) = 39 (unsigned)
- $0xD9 = -39$ (signed) = 217 (unsigned)
- $0x7F = 127$ (signed) = 127 (unsigned)
- $0x81 = -127$ (signed) = 129 (unsigned)

$$T_{min} = -128$$

$$T_{max} = 127$$

$$U_{min} = 0$$

$$U_{max} = 255$$

❖ 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

- ❖ 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 ($x \ll n$) bit vector x by n positions
 - Fill with 0's on right
- ❖ Right shift ($x \gg n$) 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
	$x \ll 3$	0001 0000
logical:	$x \gg 2$	0000 1000
arithmetic:	$x \gg 2$	0000 1000

	x	1010 0010
	$x \ll 3$	0001 0000
logical:	$x \gg 2$	0010 1000
arithmetic:	$x \gg 2$	1110 1000

Shift Operations (Review)

3.14 * 100

0001
0010
0100

❖ Arithmetic:

- Left shift ($x \ll n$) is equivalent to multiply by 2^n
- Right shift ($x \gg n$) 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 $n \geq w$ (w is bit width of x) are undefined ^{behavior not guaranteed}
- **In C:** behavior of \gg is determined by the compiler
 - In gcc / C lang, depends on data type of x ^{arithmetic/logical} (signed/unsigned)
- **In Java:** logical shift is \ggg and arithmetic shift is \gg

Left Shifting 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
$L1 = x \ll 2;$	0001100100 =	100	100
$L2 = x \ll 3;$	00011001000 =	-56	200
$L3 = x \ll 4;$	000110010000 =	-112	144

signed overflow

unsigned overflow

Right Shifting 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;` `00011110` `= 30` $/4 = 7.5$

`R2u=xu>>5;` `00000111` `= 7`

rounding (down)

Right Shifting 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;` 11111110 ~~000~~ = -2 $\frac{1}{4} = -0.5$

`R2s = xu >> 5;` 11111111 ~~10000~~ = -1

rounding (down)

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: $(x \gg 16) \ \& \ 0xFF$

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: $(x \ \& \ 0xFF0000) \gg 16$

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: $(x \gg 31) \ \& \ 0x1$
 - Assuming arithmetic shift here, but this works in either case
 - Need mask to clear 1s possibly shifted in

x	0 0000001 00000010 00000011 00000100
x>>31	00000000 00000000 00000000 0000000 0
0x1	00000000 00000000 00000000 00000001
(x>>31) & 0x1	00000000 00000000 00000000 00000000

x	1 0000001 00000010 00000011 00000100
x>>31	11111111 11111111 11111111 1111111 1
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?

<code>x=!!123</code>	00000000 00000000 00000000 00000000 1
<code>x<<31</code>	1 00000000 00000000 00000000 00000000
<code>(x<<31)>>31</code>	11111111 11111111 11111111 11111111
<code>!x</code>	00000000 00000000 00000000 00000000 0
<code>!x<<31</code>	0 00000000 00000000 00000000 00000000
<code>(!x<<31)>>31</code>	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);`