Integers II
CSE 351 Autumn 2017

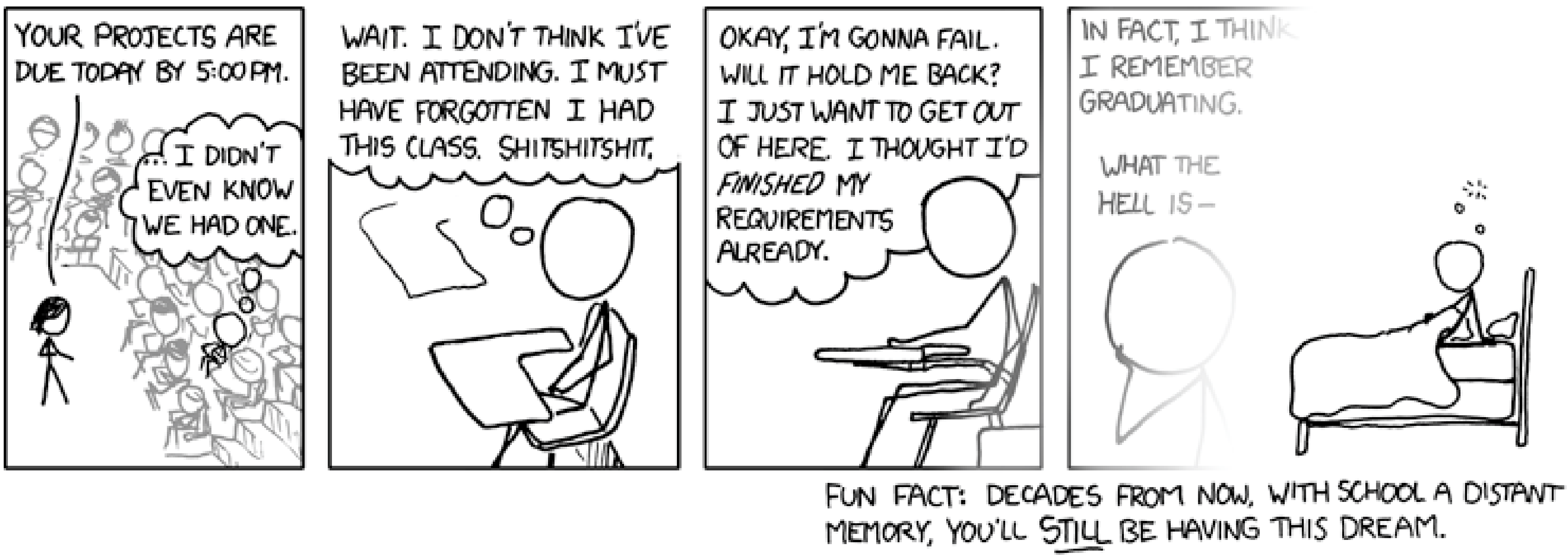
Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton Michael Zhang Parker DeWilde Ryan Wong

Sam Gehman Sam Wolfson Savanna Yee Vinny Palaniappan



http://xkcd.com/557/

Administrivia

- Lab 1 due next Friday (10/13)
  - Prelim submission (3+ of `bits.c`) due on Monday (10/9)
  - Bonus slides at the end of today's lecture have relevant examples

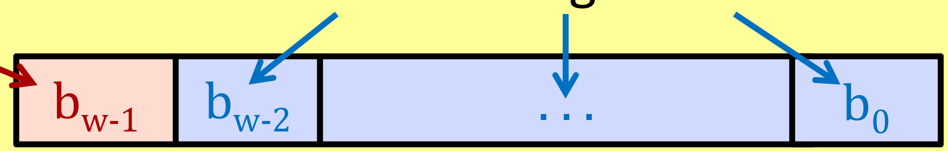
Integers

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

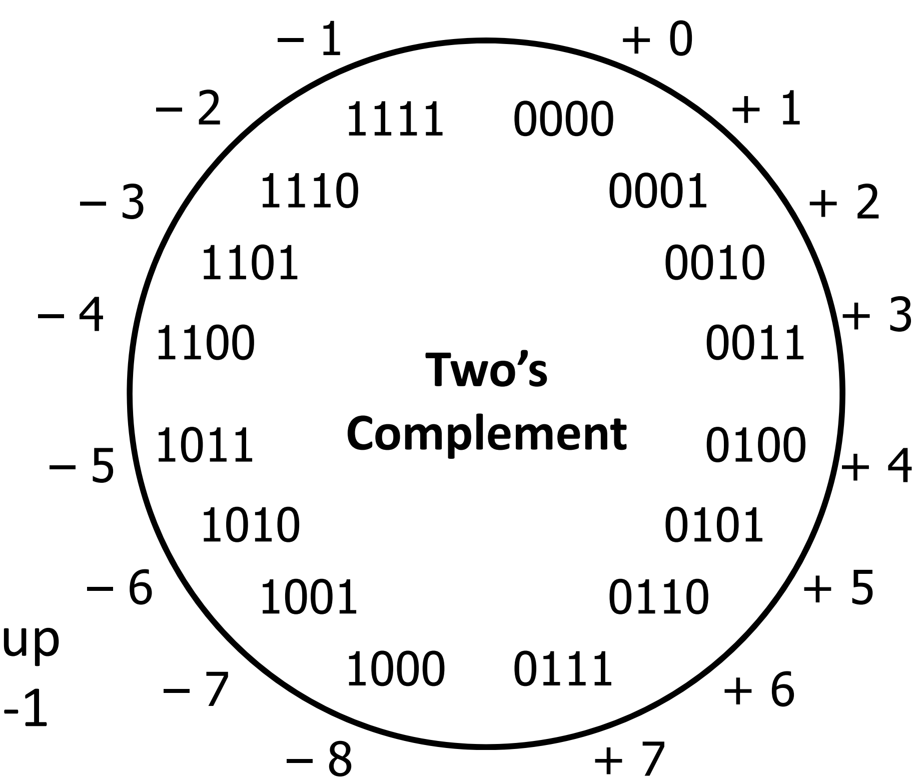
Two's Complement Negatives

- 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



Peer Instruction Question

MSB

- Take the 4-bit number encoding **x = 0b1011**
- Which of the following numbers is NOT a valid interpretation of x using any of the number representation schemes discussed last lecture?
  - Unsigned, Sign and Magnitude, Two's Complement
  - Vote at http://PollEv.com/justinh

$\boxed{\text{A. } -4}$

~~B. -5~~

~~C. 11~~

~~D. -3~~

E. We're lost…

unsigned: $8+2+1=11$

sign + mag: $1011 \rightarrow -(2+1)=-3$

two's: $-8+2+1=-5$

$-x = 0b\ 0100+1 = 5 \rightarrow x=-5$

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:**

-8+4=-4

4 +3	0100 +0011	-4 +3	1100 +0011	4 -3	1 0100 +1101
=7 ✓	0111	=-1 ✓	1111	=1 ✓	~~1~~0001

Why Does Two's Complement Work?

- For all representable positive integers x, we want:

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

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

```
   00000001        00000010         1111111
   11111111        11111110        11000011
+  ????????     +  ????????         00111101
 ----------      ----------     +  ????????
 1 00000000      1 00000000      ----------
                                 1 00000000
```

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)

0b1…1

$x + (\sim x) =$ all one's

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

$x + (\sim x + 1) = 0$

$\boxed{-x = \sim x + 1}$

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

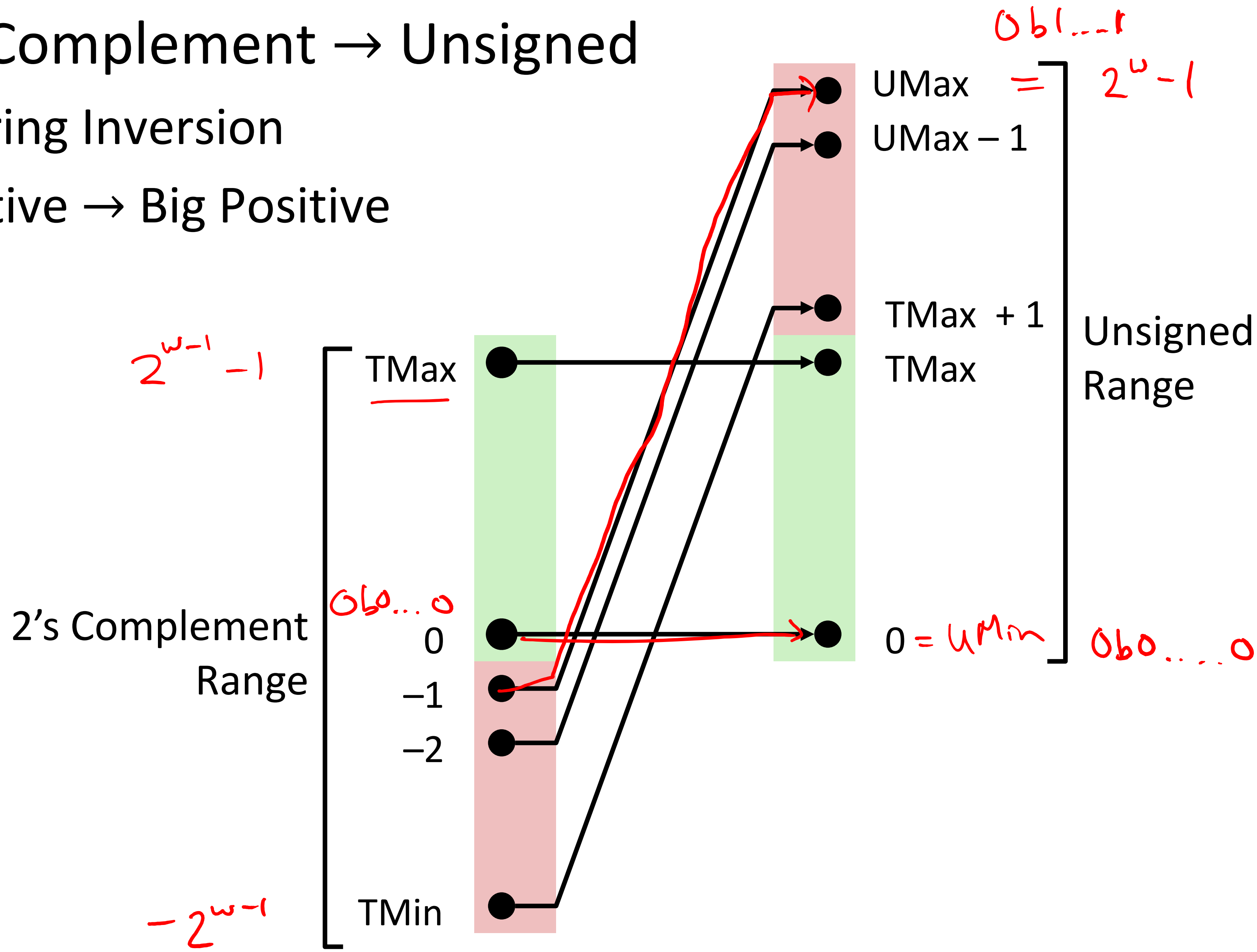
```
   00000001        00000010        11000011
+  11111111     +  11111110     +  00111101
 ----------      ----------      ----------
  100000000       100000000       100000000
```

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

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 <u>unsigned</u>**
  - 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	unsigned
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	signed
-1 1111 1111 1111 1111 1111 1111 1111 1111	>	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	>	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	signed
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	unsigned
-1 1111 1111 1111 1111 1111 1111 1111 1111	>	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	signed
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	>	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	>	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	signed

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

  UMin – UMax
  TMin – TMax

  - 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!

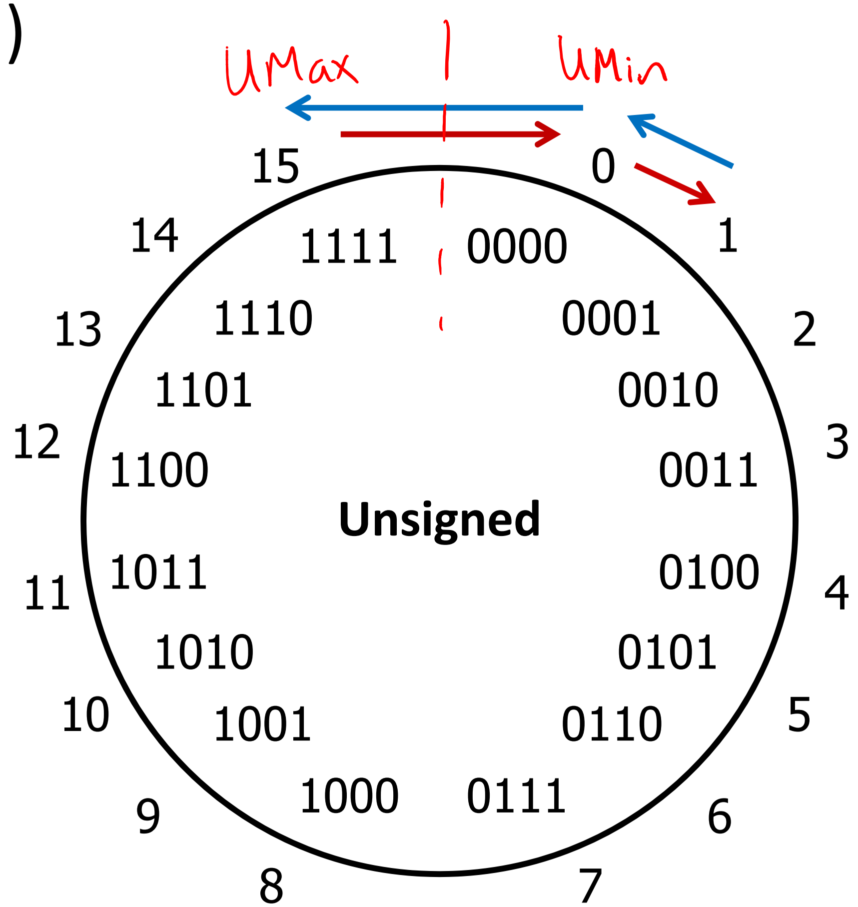
Overflow: Unsigned

- **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 ($+2^N$)

$$\begin{array}{rr} & 1 \\ - & 2 \\ \hline & \cancel{-1} \\ & 15 \end{array} \qquad \begin{array}{rr} & \cancel{1}\overset{2}{0}\overset{\cancel{2}}{0}\overset{2}{0}1 \\ - & 0010 \\ \hline & 1111 \end{array}$$



$\pm 2^N$ because of modular arithmetic

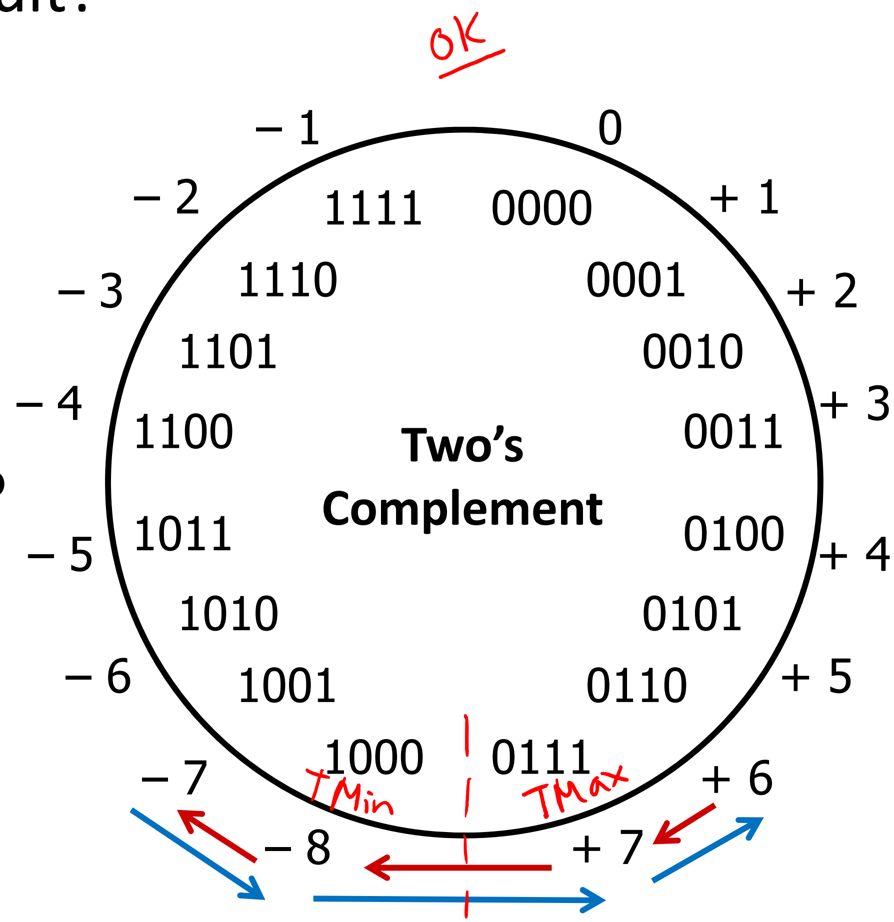
Overflow: Two's Complement

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

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

- **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`
    1 byte 2 bytes 4 bytes 8 bytes
- **4-bit → 8-bit Example:**
  - Positive Case
    - ✓ Add 0's?

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

Peer Instruction Question

- Which of the following 8-bit numbers has the same *signed* value as the 4-bit number **0b1100**? → $-2^3+2^2=-4$

  $-x = $ 0b 0011 $+1$ = 4 → $x=-4$
  - Underlined digit = MSB
  - Vote at http://PollEv.com/justinh

~~A.~~ 0b 0000 1100 — positive number

~~B.~~ 0b 1000 1100 — much too negative: $-2^7+2^3+2^2=-116$

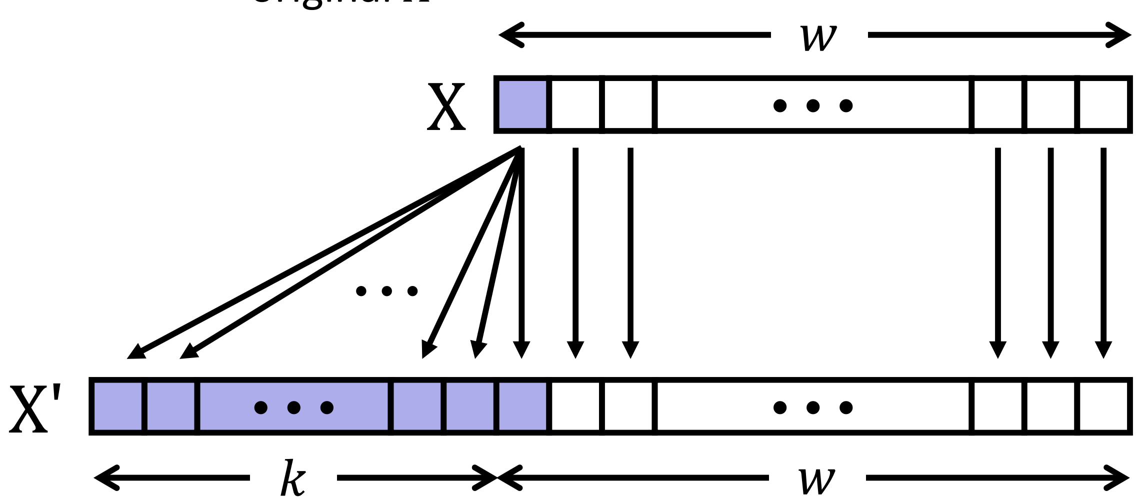
C. 0b 1111 1100 — $-C = $ 0b0000 0011 $+1$ = 4

D. 0b 1100 1100 — $-D = $ 0b0011 0011 $+1$ = $32+16+4 = 52$

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



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

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<<n`) bit vector `x` by `n` positions
  - Throw away (drop) extra bits on left
  - Fill with `0`s on right
- Right shift (`x>>n`) bit-vector `x` by `n` positions
  - Throw away (drop) extra bits on right
  - 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`

Shift Operations

- Left shift (x<<n)
  - Fill with 0s on right
- Right shift (x>>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≥w (bit width of x) are *undefined*
  - **In C:** behavior of >> is determined by compiler
    - In gcc / C lang, depends on data type of x (signed/unsigned)
  - **In Java:** logical shift is >>> and arithmetic shift is >>

8-bit number

	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

Shifting Arithmetic?

- What are the following computing?
  - x>>n
    ```
           4               2
    0b 0100 >> 1 = 0b 0010
           4               1
    0b 0100 >> 2 = 0b 0001
    ```
    - Divide by 2^n
  - x<<n
    ```
           1               2
    0b 0001 << 1 = 0b 0010
           1               4
    0b 0001 << 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
`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

$200 - 256$, $256 \rightarrow 2^8$

signed overflow

$400 - 256$, $256 \rightarrow 2^8$

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;  00011110~~000~~ =  30
R2u=xu>>5;  00000111~~10000~~ =   7
```

$240/8 = 30$

$30/4 = 7.5$

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; 11111110~~000~~ = -2 $/4 = -0.5$

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

rounding (down)

Peer Instruction Question

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. Compare with your neighbor(s)!

- Assume we are using 8-bit arithmetic:

Expression	Example:	General:
`x == (unsigned char) x` (unsigned)	x = 0	works for all x
`x >= 128U` (unsigned) 0b1000 0000	x = -1	any x < 0
`x != (x>>2)<<2`	x = 3	any x where lowest two bits are not 0b00
`x == -x` • Hint: there are two solutions	x = 0	① x = 0b0...0 = 0 ② x = 0b10...0 = -128
`(x < 128U) && (x > 0x3F)`	x-64	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 1.

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

- 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 1s 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);`