Administrivia

- hw4 due 10/12, hw5 due 10/14

- Lab 1a due Monday (10/12)
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope

- Lab 1b released tomorrow, due 10/19
  - 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*!
  - Hide 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:**
  - $\text{UMin}, \text{UMax}, \text{TMin}, \text{TMax}$
  - 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 $T_{\text{min}}$ for a fictional 6-bit wide integer data type?

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

- What is the result of the following expressions?
  - `(signed char)uc >> 2`
  - `(unsigned char)uc >> 3`
Why Does Two’s Complement Work?

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

\[
\begin{align*}
\text{bit representation of } x &+ \text{bit representation of } \neg x \\
&= 0 \quad \text{(ignoring the carry-out bit)}
\end{align*}
\]

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

\[
\begin{align*}
00000001 &+ ???????? & 00000010 &+ ???????? & 11000011 &+ ???????? \\
00000000 &+ 00000000 & 00000000 &+ 00000000 & 00000000 &+ 00000000
\end{align*}
\]
Why Does Two’s Complement Work?

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

  \[
  \begin{align*}
  \text{bit representation of } x \\
  + \text{bit representation of } -x \\
  0 \quad \text{(ignoring the carry-out bit)}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  00000001 & \quad 00000010 & \quad 11000011 \\
  + 11111111 & + 11111110 & + 00111101 \\
  100000000 & 100000000 & 100000000
  \end{align*}
  \]

  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 → Unsigned
  - Ordering Inversion
  - Negative → Big Positive
Values To Remember

- **Unsigned Values**
  - $U_{\text{Min}} = 0b00...0 = 0$
  - $U_{\text{Max}} = 0b11...1 = 2^w - 1$

- **Two’s Complement Values**
  - $T_{\text{Min}} = 0b10...0 = -2^{w-1}$
  - $T_{\text{Max}} = 0b01...1 = 2^{w-1} - 1$
  - $-1 = 0b11...1$

- **Example: Values for $w = 64$**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>18,446,744,073,709,551,615</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>9,223,372,036,854,775,807</td>
<td>7F FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-9,223,372,036,854,775,808</td>
<td>80 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
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 `<, >, ==, <=, >=`
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
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

- **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
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$

![Diagram](image.png)
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 \( mod 2^w \)
Arithmetic Overflow

- 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!

<table>
<thead>
<tr>
<th>Bits</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

- **Addition:** drop carry bit ($-2^N$)
  
  \[
  \begin{array}{c}
  15 \\
  + \ 2 \\
  \hline
  17 \\
  \end{array}
  \quad \quad \quad \quad
  \begin{array}{c}
  1111 \\
  + \ 0010 \\
  \hline
  10001 \\
  \end{array}
  
  1

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

  \[
  \begin{array}{c}
  1 \\
  - \ 2 \\
  \hline
  -1 \\
  \end{array}
  \quad \quad \quad \quad
  \begin{array}{c}
  10001 \\
  - \ 0010 \\
  \hline
  1111 \\
  \end{array}
  
  15

$\pm 2^N$ because of modular arithmetic
Overflow: Two’s Complement

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

  $$
  \begin{array}{c}
  6 \\
  + 3 \\
  \hline
  9
  \end{array}
  \quad \begin{array}{c}
  0110 \\
  + 0011 \\
  \hline
  1001
  \end{array}
  \quad \text{result?}
  $$

- **Subtraction:** $(-)+(-)=(+)$?

  $$
  \begin{array}{c}
  -7 \\
  - 3 \\
  \hline
  -10
  \end{array}
  \quad \begin{array}{c}
  1001 \\
  - 0011 \\
  \hline
  0110
  \end{array}
  $$

For signed: overflow if operands have same sign and result’s sign is different
Practice Questions 2

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

- For the following additions, did signed and/or unsigned overflow occur?
  - \(0x27 + 0x81\)
  - \(0x7F + 0xD9\)
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

- 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 \(x\))

\[
\begin{array}{c|c}
\text{x} & 0010 0010 \\
\text{x\ll3} & 0001 0000 \\
\text{x\gg2} & 0000 1000 \\
\text{logical:} & \text{arithmetic:}
\end{array}
\quad
\begin{array}{c|c}
\text{x} & 1010 0010 \\
\text{x\ll3} & 0001 0000 \\
\text{x\gg2} & 0010 1000 \\
\text{logical:} & \text{arithmetic:}
\end{array}
\]
Shift Operations

- **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!

- **Notes:**
  - Shifts by \(n < 0\) or \(n \geq w\) (\(w\) is bit width of \(x\)) are **undefined**
  - **In C:** behavior of \(>>\) is determined by the compiler
    - In gcc / C lang, depends on data type of \(x\) (signed/unsigned)
  - **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: \( x \times 2^n \)?

\[
\begin{align*}
\text{x} &= 25; \quad 00011001 = 25 & \quad \text{Signed} & \quad 25 \quad \text{Unsigned} & \quad 25 \\
L1 &= \text{x}<<2; \quad 0001100100 = 100 & \quad \text{Signed} & \quad 100 \quad \text{Unsigned} & \quad 100 \\
L2 &= \text{x}<<3; \quad 00011001000 = -56 & \quad \text{Signed} & \quad 200 \quad \text{Unsigned} & \quad 200 \\
L3 &= \text{x}<<4; \quad 000110010000 = -112 & \quad \text{Signed} & \quad 144 \quad \text{Unsigned} & \quad 144
\end{align*}
\]
Right Shifting Arithmetic 8-bit Examples

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

\[
xu = 240u; \quad 11110000 = 240
\]
\[
R1u = xu >> 3; \quad 00011110000 = 30
\]
\[
R2u = xu >> 5; \quad 0000011110000 = 7
\]

(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: \( x / 2^n \)?

\[
x_s = -16; \quad 11110000 = -16
\]
\[
R1s=xu>>3; \quad 111111110000 = -2
\]
\[
R2s=xu>>5; \quad 1111111110000 = -1
\]
Challenge 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 arithmetic:
  - \( x == \) (unsigned char) \( x \)
  - \( x >= 128U \)
  - \( x !== (x>>2)<<(2) \)
  - \( x == -x \)
    - 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
Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1b.

- Extract the 2\textsuperscript{nd} most significant byte of an \texttt{int}
- Extract the sign bit of a signed \texttt{int}
- Conditionals as Boolean expressions
Using Shifts and Masks

- Extract the 2\textsuperscript{nd} most significant \textit{byte} of an \texttt{int}:
  - First shift, then mask: \((x>>16) \& 0xFF\)
  - Or first mask, then shift: \((x \& 0xFF0000) >> 16\)
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

<table>
<thead>
<tr>
<th>$x$</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &gt;&gt; 31$</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>$0x1$</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>$(x &gt;&gt; 31) &amp; 0x1$</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>10000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &gt;&gt; 31$</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>$0x1$</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>$(x &gt;&gt; 31) &amp; 0x1$</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- Conditionals as Boolean expressions
  - For int \( x \), what does \( (x\ll31)\gg31 \) do?

| \( x = !!123 \) | 00000000 00000000 00000000 00000001 |
|\( x\ll31 \) | 10000000 00000000 00000000 00000000 |
|\( (x\ll31)\gg31 \) | 11111111 11111111 11111111 11111111 |
| \( !x \) | 00000000 00000000 00000000 00000000 |
| \( !x\ll31 \) | 00000000 00000000 00000000 00000000 |
|\( (!x\ll31)\gg31 \) | 00000000 00000000 00000000 00000000 |

- Can use in place of conditional:
  - In C: \( \text{if}(x) \{ \text{a}=y; \} \ \text{else} \ \{ \text{a}=z; \} \) equivalent to \( \text{a}=x?y:z; \)
  - \( a=((x\ll31)\gg31)\&y) \mid ((!x\ll31)\gg31)\&z; \)