Integers II, Floating Point I
CSE 351 Summer 2021

Instructor: Mara Kirdani-Ryan
Teaching Assistants: Kashish Aggarwal, Nick Durand, Colton Jobes, Tim Mandzyuk

http://xkcd.com/1953/
Gentle and Loving Reminders!

- Hw4, hw5, lab1a due Monday 7/2 – 8pm
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope for lab1a
- hw6 and hw7 due Friday 7/9 – 8pm
- Lab 1b released today, due 7/9
  - Bit manipulation problems using custom data type
  - Some helpful examples in the bonus slides today and next time
- Anything else?
  - We’re here to help! You just need to ask!
Gradescope Lab Turnin

- Make sure you pass the File and Compilation Check!

- Doesn’t indicate if you passed all tests, just indicates that all the correct files were found and there were no compilation or runtime errors.

- Use the testing programs we provide to check your solution for correctness (on attu or the VM)
Ohyay’s more secure

- You’ll need an invite to join, anyone on staff can give invites
  - This shouldn’t affect anyone, let us know if it does!
Quick Aside: C Macros

- You’ll use C macros in Lab1b for bit masks
- Syntax is of the form:
  
  ```
  #define NAME expression
  ```
- Can use “NAME” instead of “expression” in code
- Useful to help with readability/factoring in code
  - Especially useful for defining constants such as bit masks!
Quick Aside: C Macros

- Are NOT exactly the same as a constant in Java
  - Does naïve copy and replace *before* compilation.
  - Everywhere the characters “NAME” appear in the code, the characters “expression” will now appear instead.
  - Lots of gotchas, be careful!
- See Lecture 4 (Integers I) slides for examples
How are y’all feeling today?

We’re going to talk about automation!
Breakouts!

What work is being automated? Why? Is there anything that separates “automated work” from work still performed by people?
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and **arithmetic operations** – for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
Learning Objectives

By the end of this lecture, you should be able to:

- Explain integer overflow at the level of bits (ideally in plain language to someone outside this class)
- Determine when signed/unsigned overflow will occur
- Convert decimal numbers to fixed point encoding
- Explain the values embedded in the first computers, with comparisons to computing today
Reading Review

- **Terminology:**
  - UMin, UMax, Tmin, 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
Review Questions

- What is the value (and encoding) of $\text{TMin}$ for a fictional 6-bit wide integer data type?

- For `unsigned char uc = 0xA1;`, what is the result of the cast `(short)uc`?

- What is the result of the following expressions?
  - `(signed char)uc >> 2`
  - `(unsigned char)uc >> 3`
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 highest carry bit
    - Called modular addition: result is sum \( \text{modulo } 2^w \)

- **4-bit Examples:**

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>+0011</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>0111</td>
</tr>
</tbody>
</table>
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 highest carry bit
    - Called modular addition: result is sum \( \text{modulo } 2^w \)

- 4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>
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 highest carry bit
    - Called modular addition: result is sum \( \text{modulo } 2^w \)

- 4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>=0111</td>
<td>= 7</td>
</tr>
</tbody>
</table>
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 highest carry bit*
    - Called modular addition: result is sum *modulo* $2^w$

- 4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>- 4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>=0111</td>
<td>= 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>+0011</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>+1101</td>
<td></td>
</tr>
<tr>
<td>=0001</td>
<td></td>
</tr>
</tbody>
</table>
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 highest carry bit
    - Called modular addition: result is sum \( \text{modulo } 2^w \)
- 4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>-4</td>
<td>1100</td>
<td>-4</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
<td>+3</td>
<td>+1101</td>
<td></td>
</tr>
<tr>
<td>=0111</td>
<td>=7</td>
<td>=1111</td>
<td>=-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 highest carry bit
    - Called modular addition: result is sum \( \mod 2^w \)

- **4-bit Examples:**

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>-4</td>
<td>1100</td>
<td>-4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
<td>+3</td>
<td>+1101</td>
<td>-3</td>
</tr>
<tr>
<td>=0111</td>
<td>=7</td>
<td>=1111</td>
<td>=-1</td>
<td>=0001</td>
<td>=1</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers \( x \), we want:
  
  
  \[
  \text{bit representation of } \ x \\
  + \text{bit representation of } -x
  \]

  (ignoring the carry-out bit)

  \[
  0
  \]

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

  
  
  \[
  \begin{array}{c}
  00000001 \\
  \hline
  00000000
  \end{array}
  \]

  
  
  \[
  \begin{array}{c}
  00000010 \\
  \hline
  00000000
  \end{array}
  \]

  
  
  \[
  \begin{array}{c}
  11000011 \\
  \hline
  00000000
  \end{array}
  \]
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want: 
  
  $\text{bit representation of } x + \text{bit representation of } -x$ 
  
  (ignoring the carry-out bit)

  
  $0$

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

<table>
<thead>
<tr>
<th>Bit Representation</th>
<th>Bit Representation</th>
<th>Bit Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001 + 11111111</td>
<td>00000010 + 11111110</td>
<td>11000011 + 00111101</td>
</tr>
</tbody>
</table>

These are the bitwise complement plus 1!

$-x == \sim x + 1$
Signed/Unsigned Conversion Visualized

- Two’s Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

Unsigned Range

0/UMin

TMin

0

0

TMax

UMax

UMax – 1

TMax + 1

TMax

0/UMin
Values To Remember (or lookup)

- **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 \(w = 64\)

<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 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</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – *Lab 1a*
- **In C: Signed, Unsigned and Casting**
- Consequences of finite width representations
  - Overflow, sign extension
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 *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - Examples: 0U, 4294967259u

- Expression Evaluation
  - Mixing unsigned and signed in an expression means *signed values are implicitly cast to unsigned*
  - Including comparison operators <, >, ==, <=, >=
# Casting Surprises

- **True**
- **False**
- 😶 Help!

## 32-bit examples:

- $T_{\text{Min}} = -2,147,483,648$, $T_{\text{Max}} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int) 2147483648U</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises

- **True**
- **False**
- **Help!**

### 32-bit examples:

- **TMin = -2,147,483,648, Tmax = 2,147,483,647**

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 0000 0000 0000 0000 0000 0000</td>
<td>&lt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0 0000 0000 0000 0000 0000 0000 0000</td>
<td>T</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>2147483648U 1000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>(int) 2147483648U 1000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises

- True
- False
- Help!

32-bit examples:

- \( T_{\text{Min}} = -2,147,483,648, \ T_{\text{Max}} = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&lt;</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>T</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647U</td>
<td></td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td>T</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td>T</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648U</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
</tbody>
</table>
Casting Surprises

- **True**
- **False**
- **Help!**

- **32-bit examples:**

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

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>T</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>F</td>
</tr>
<tr>
<td>2147483647U 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>T</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>2147483648U 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>(int) 2147483648U 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
## Casting Surprises

- **True**
- **False**
- **Help!**

### 32-bit examples:

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

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>T</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&lt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int) 2147483648U</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises

- 💛 True
- 💚 False
- 😬 Help!

32-bit examples:

- Tmin = -2,147,483,648, Tmax = 2,147,483,647

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647 U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>T</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&lt;</td>
<td>-2</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648 U</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int) 2147483648 U</td>
<td>F</td>
</tr>
</tbody>
</table>
Casting Surprises

- ♡ True
- 💚 False
- 😶 Help!

32-bit examples:

- $T_{\text{Min}} = -2,147,483,648$, $T_{\text{Max}} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>T</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&lt;</td>
<td>-2</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>T</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int) 2147483648U</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises

- ❤️ True
- 🌶️ False
- 😷 Help!

32-bit examples:

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

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>F</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>T</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2</td>
<td>T</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&lt;</td>
<td>-2</td>
<td>F</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>2147483648</td>
<td>T</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int) 2147483648U</td>
<td>F</td>
</tr>
</tbody>
</table>
How do we feel about casting?
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – for Lab 1a
- In C: Signed, Unsigned and Casting
- **Consequences of finite width representations**
  - Overflow, sign extension, limits
## 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!

### Bit Encoding

<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>0</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

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

  \[
  \begin{array}{c}
  15 \\
  + \ 2 \\
  \hline
  17
  \end{array}
  \begin{array}{c}
  1111 \\
  + \ 0010 \\
  \hline
  10001
  \end{array}
  \]

  \[
  \begin{array}{c}
  1 \\
  - \ 1 \\
  \hline
  15
  \end{array}
  \begin{array}{c}
  0001 \\
  - \ 0010 \\
  \hline
  1111
  \end{array}
  \]

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

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

\[\pm 2^N \text{ because of modular arithmetic}\]
Overflow: Two’s Complement

- **Addition:** $(+) + (+) = (−)$ result?
  - $6 + 3 = 9$
  - $0110 + 0011 = 1001$
- **Subtraction:** $(−) + (−) = (+)$?
  - $−7 + −3 = −10$
  - $1001 + −0011 = 0110$

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

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

- For the following additions, did signed and/or unsigned overflow occur?
  - \(0x27 + 0x81\)
  - \(0x7F + 0xD9\)
Practice Questions 2

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

- For the following additions, did signed and/or unsigned overflow occur?
  - 💚 True
  - 💚 False
  - 😱 Help!
  - 0x27 + 0x81
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`

- Negative Case?
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
- $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_1, x_0$

$k$ copies of MSB
original $X$

![Diagram of sign extension](image-url)
Sign Extension Example

- Convert from smaller to larger integral data types
- C automatically performs sign extension
  - Java too

```java
short int x = 12345;
int ix = (int) x;
short int y = -12345;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>30 39</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 00111001</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF CF C7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>
How are we feeling about integers?
Aside: Unsigned Multiplication in C

Operands:
\[ w \text{ bits} \]

\[ u \]

\[ \ast \]

\[ v \]

True Product:
\[ 2w \text{ bits} \]

\[ u \cdot v \]

Discard \( w \) bits:
\[ w \text{ bits} \]

\[ \text{UMult}_w (u, v) \]

- 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:
Aside: Multiplication w/ shift & add

- **Operation** \( u << k \) gives \( u \times 2^k \)
  - Both signed and unsigned

**Operands:** \( w \) bits

<table>
<thead>
<tr>
<th></th>
<th>( u )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \times 2^k )</td>
<td>( 0 \cdots 010 \cdots 00 )</td>
</tr>
</tbody>
</table>

**True Product:** \( w + k \) bits

<table>
<thead>
<tr>
<th></th>
<th>( u \cdot 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discard ( k ) bits: ( w ) bits</td>
<td>( \bullet \cdots \bullet \cdots 0 \cdots 0 )</td>
</tr>
</tbody>
</table>

**Examples:**

- \( u << 3 \) == \( u \times 8 \)
- \( u << 5 \) - \( u << 3 \) == \( u \times 24 \)
- Most machines shift and add faster than multiply
  - **Compiler generates this code automatically**
Number Representation Revisited

- What can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII, Unicode)
  - Addresses

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. $6.02 \times 10^{23}$)
  - Very small numbers (e.g. $6.626 \times 10^{-34}$)
  - Special numbers (e.g. $\infty$, NaN)
Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page “standard”…
Representation of Fractions

- Decimal Point (B=10)

Example 6-digit representation:

```
xx. yyyy
```

```
B^1 \quad B^0 \quad B^{-1} \quad B^{-2} \quad B^{-3} \quad B^{-4}
```
Representation of Fractions

- “Binary Point,” like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation:

- **Example:** $10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$
Fractional Binary Numbers

- Representation
  - Bits to right of “binary point” == fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \cdot 2^k$
Fractional Binary Numbers

- **Value Representation**
  - 5 and 3/4: $101.11_2$
  - 2 and 7/8: $10.111_2$
  - 47/64: $0.101111_2$

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$
Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form \( x \times 2^y \) (y can be negative)
  - Other rational numbers have repeating bit representations

### Value: Binary Representation:

- \( \frac{1}{3} = 0.333333..._{10} = 0.01010101[01]..._2 \)
- \( \frac{1}{5} = 0.2 = 0.001100110011[0011]..._2 \)
- \( \frac{1}{10} = 0.1 = 0.0001100110011[0011]..._2 \)
Fixed Point Representation

- Implied binary point. Two example schemes:
  
  1. The binary point is between bits 2 and 3
     \[ \text{b}_7 \text{b}_6 \text{b}_5 \text{b}_4 \text{b}_3 \text{[.]} \text{b}_2 \text{b}_1 \text{b}_0 \]
  
  2. The binary point is between bits 4 and 5
     \[ \text{b}_7 \text{b}_6 \text{b}_5 \text{[.]} \text{b}_4 \text{b}_3 \text{b}_2 \text{b}_1 \text{b}_0 \]

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have.

- Fixed point = fixed range and fixed precision
  
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

- Hard to pick how much you need of each!
Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but $1.2 \times 10^7$ In C: 1.2e7
    - Not 0.0000012, but $1.2 \times 10^{-6}$ In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but $1.1 \times 2^4$
    - Not 0.000101, but $1.01 \times 2^{-4}$
- We have to divvy up the bits we have among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent
Scientific Notation (Decimal)

- **Normalized form**: exactly one digit (non-zero) to left of decimal point

- Alternatives to representing $1/1,000,000,000,000$
  - Normalized: $1.0 \times 10^{-9}$
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$
Scientific Notation (Binary)

- Computer arithmetic that supports this called **floating point** due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)
Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: \(1.011_2 \times 2^4 = 10110_2 = 22_{10}\)
    - Example: \(1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}\)

- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: \(1101.001_2 = 1.101001_2 \times 2^3\)

- Practice: Convert \(11.375_{10}\) to normalized binary scientific notation
  \[1.011011 \times 2^3\]
How are we feeling about fixed point?
Technical 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 much in \( w \) bits
  - When we exceed the limits, *arithmetic overflow* occurs
  - *Sign extension* tries to preserve value when expanding
- Floating point approximates real numbers
  - More details on Friday!
What values were embedded in the first computing machine?
(Modern) Hardware: Historic View

- **Computer**: one who computes

![Image of the women of Bletchley Park, Credit: BBC]

- Mostly white cis-women
- “Boring, repetitive work”, doing math quickly
But, let’s go further back!
The “Father” of Computing (~1820)
The “Father” of Computing (~1820)

Charles Babbage

Ada Lovelace
The “Father” of Computing (~1820)

“Great” man

First Programmer!
The “Father” of Computing (~1820)

“Great” man

“Great” machine
Why make an analytical engine?
Why make an analytical engine?

Let’s go further back!
The Division of Labor

“The Wealth of Nations”, 1776
The Division of Labor (Adam Smith)

“The Wealth of Nations”, 1776
Making Butter

1. Milk cows
2. Turn milk into cream
3. Ferment cream
4. Churn cream
5. Strain milk
6. Get butter!

- Historically, all one person
- Adam Smith: it’ll go faster if we specialize
- “Division of Labor”

Basically, specialization improves efficiency
Inspiring Babbage

Cool, I guess.
Babbage, inspo by....
Babbage, inspo by Gaspard De Prony

- Applied division of labor to produce logarithms!
Who wants butter, when you can have logarithms?
Babbage, inspo by Gaspard De Prony

- Applied division of labor to produce logarithmic tables
- “…manufacture logarithms as one manufactures pins”
- 5 experts, 8 managers, 70 human computers
But what if we could automate the human computers?
The “Father” of Computing (~1820)

“Great” man

“Great” machine
But what if we could automate the human computers?

What about the hairdressers, turned human computers?

We’ve seen this line of thinking before!
Automation/Augmentation

- “Boring, repetitive work” should be automated
  - This tends to eliminate jobs for marginalized folks
- ”Boring, repetitive work” is robot work
  - Performed by non-humans, or ”less humans”
- Thematic through history of computing
  - Even going back to ”the father of computing”
  - Of course, this includes modern incarnations
  - Computing has been “automating” for a while!
Why are we automating?

- Efficiency of labor – robots don’t need healthcare or time off, and are frequently faster
  - Industrialized mechanization: efficiency for profit
- Efficiency of conflict – robots can kill remotely
  - Also, quicker calculation of ballistics, other math
  - Militarization: efficiency for safety (i.e. win faster)

If anyone’s studied Marx, feel free to chime in!
Charles Babbage

- Credits De Prony with inspiration
- Coins *Babbage Principle*
  - "High Skilled" workers regularly perform tasks below their abilities, yet are paid a “high-skill” wage. We should effectively divide labor so that no one paid a “high-skill” wage was doing “low-skill” work.
- Does this sound familiar to anyone?
CS views on labor

- What work does CS consider “high skill”?
  - Programming, obv.
  - Others?
- What work does CS consider “low skill”?
What values were embedded in the first computing machine?
Efficiency!
Computing tends to value efficiency over humanity!

The Hardware/Software Interface consistently reflects this!
A reminder: your self-worth isn’t tied to your efficiency!
Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1.

- 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 2nd most significant byte of an int:
  - First shift, then mask: \((x >> 16) \& 0xFF\)

<table>
<thead>
<tr>
<th></th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>00000000 00000000 00000001 00000100</td>
</tr>
<tr>
<td>x&gt;&gt;16</td>
<td>00000000 00000000 00000001 11111111</td>
</tr>
<tr>
<td>0xFF</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td>(x&gt;&gt;16) &amp; 0xFF</td>
<td>00000000 00000000 00000000 00000100</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>00000001 00000010 00000011 00000100</td>
</tr>
<tr>
<td>0xFF0000</td>
<td>00000000 11111111 00000000 00000000</td>
</tr>
<tr>
<td>x &amp; 0xFF0000</td>
<td>00000000 00000010 00000000 00000000</td>
</tr>
<tr>
<td>(x&amp;0xFF0000) &gt;&gt;16</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
</tbody>
</table>
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 00000001</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<<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);`