Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-1](https://tinyurl.com/CSE351-7-1)
- hw4 and hw5 due Monday 7/6 – 10:30am
- hw6 and hw7 due Friday 7/10 – 10:30am
  - Will post Monday’s slides later today so you can get started
- Lab 1a due Monday (7/6) *(try to finish by Friday!)*
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b released tomorrow, due 7/10
  - Bit manipulation problems using custom data type
  - Today’s bonus slides have helpful examples, tomorrow’s section will have helpful examples too
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)
Quick Aside: C Macros

- Lab1b will have you use some C macros for bit masks
- Syntax is of the form:
  
  ```
  #define NAME expression
  ```
- Can now use “NAME” instead of “expression” in code
- Useful to help with readability/factoring in code
  - Especially useful for defining constants such as bit masks!
- 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.
- See Lecture 4 (Integers I) slides for example usages
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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 \( \text{mod} \ 2^w \)

- **4-bit Examples:**
  
  \[
  \begin{array}{c|c}
  \text{HW} & \text{TC} \\
  \hline
  0100 & +0011 \\
  \hline
  = & \\
  \end{array}
  \]
  
  \[
  \begin{array}{c|c}
  \text{HW} & \text{TC} \\
  \hline
  1100 & +0011 \\
  \hline
  = & \\
  \end{array}
  \]
  
  \[
  \begin{array}{c|c}
  \text{HW} & \text{TC} \\
  \hline
  0100 & +1101 \\
  \hline
  = & \\
  \end{array}
  \]
Why Does Two’s Complement Work?

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

\[
\text{bit representation of } x + \text{bit representation of } -x + 0 \text{ (ignoring the carry-out bit)}
\]

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

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

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

\[
\text{bit representation of } x \\
+ \text{bit representation of } -x \\
= \text{0 (ignoring the carry-out bit)}
\]

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

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

These are the bitwise complement plus 1!

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

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

![Diagram showing the conversion between two's complement and unsigned numbers. The diagram illustrates the range of values for both types and how negative values are mapped to large positive values in the unsigned format.](image)
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 $w = 64$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMax</strong></td>
<td>18,446,744,073,709,551,615</td>
<td>FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td><strong>TMax</strong></td>
<td>9,223,372,036,854,775,807</td>
<td>7F FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td><strong>TMin</strong></td>
<td>-9,223,372,036,854,775,808</td>
<td>80 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</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed

- Shifting and arithmetic operations – useful for 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 considered \textit{signed} integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force \textit{unsigned}
    - Examples: \texttt{0U, 4294967259u}

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then \textit{signed values are implicitly cast to unsigned} (\texttt{"dominates"})
  - Including comparison operators \texttt{<, >, ==, <=, >=}
## Casting Surprises

- **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></td>
<td>0U</td>
<td></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></td>
<td>0</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>-1</td>
<td></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></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></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></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></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></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></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>
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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 \((-2^N)\)

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

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

\[
\begin{array}{c}
1 \\
\underline{- 2} \\
\hline
-1 \\
\text{Blue line}
\end{array}
\quad
\begin{array}{c}
10001 \\
\underline{- 0010} \\
\hline
1111 \\
\text{Red line}
\end{array}
\]

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

- **Addition:** 
  \[(+\,+)\,+(=\,)=\,(-)\,\text{result?}\]
  
  \[
  \begin{array}{c}
  6 \\
  + 3 \\
  \hline
  9 \\
  -7
  \end{array}
  \]

- **Subtraction:** 
  \[(\,-\,)+\,(\,-\,)=\,(\,+\,)?\]
  
  \[
  \begin{array}{c}
  -7 \\
  -3 \\
  \hline
  -10 \\
  6
  \end{array}
  \]

**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
  - Negative Case?
Polling Question [Int II - a]

- Which of the following 8-bit numbers has the same signed value as the 4-bit number \texttt{0b1100}?
  - Underlined digit = MSB
  - Vote at http://pollev.com/pbjones

A. \texttt{0b 0000 1100}
B. \texttt{0b 1000 1100}
C. \texttt{0b 1111 1100}
D. \texttt{0b 1100 1100}
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
- $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$

![Diagram showing sign extension process](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>
Practice Question

For the following expressions, find a value of \texttt{signed char} \( x \), if there exists one, that makes the expression \texttt{TRUE}. Compare with your neighbor(s)!

- Assume we are using 8-bit arithmetic:
  - \( x == (\texttt{unsigned char}) x \)
  - \( x >= 128U \)
  - \( x != (x>>2) \ll 2 \)
  - \( x == -x \)
    - Hint: there are two solutions
  - \( (x < 128U) \&\& (x > 0x3F) \)
Aside: Unsigned Multiplication in C

Operands:  
\(w\) bits

True Product:  
\(2w\) bits

Discard \(w\) bits:  
\(w\) bits

- Standard Multiplication Function
  - Ignores high order \(w\) bits

- Implements Modular Arithmetic
  - \(\text{UMult}_w(u, v) = u \cdot v \mod 2^w\)
Aside: Multiplication with shift and add

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

**Operands:** $w$ bits

**True Product:** $w + k$ bits

**Discard $k$ bits:** $w$ bits

- **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)
  ▪ 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

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

Example 6-bit representation:

\[ xx \cdot yyyyy \]

\[ 2^1 \] \[ 2^0 \] \[ 2^{-1} \] \[ 2^{-2} \] \[ 2^{-3} \] \[ 2^{-4} \]

- Example: \[ 10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10} \]
Representation of Fractions

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

Example 6-bit representation: \( \text{xx.yyyy} \)

\[ 2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \]

❖ In this 6-bit representation:

- What is the encoding and value of the smallest (most negative) number?
- What is the encoding and value of the largest (most positive) number?
- What is the smallest number greater than 2 that we can represent?
Fractional Binary Numbers

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

\[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Fractions Binary Numbers

- Value: 5 and 3/4, 2 and 7/8, 47/64
- Representation: 101.11₂, 10.111₂, 0.101111₂

Observations

- Shift left = multiply by power of 2
- Shift right = divide by power of 2
- Numbers of the form 0.111111...₂ are just below 1.0
  - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... → 1.0
  - Use notation 1.0 − ε
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>$0.333333..._{10}$ = 0.01010101[01]...$_2$</td>
</tr>
<tr>
<td>1/5</td>
<td>$0.001100110011[0011 ]..._2$</td>
</tr>
<tr>
<td>1/10</td>
<td>$0.0001100110011[0011 ]..._2$</td>
</tr>
</tbody>
</table>
Fixed Point Representation

- Implied binary point. Two example schemes:
  1. The binary point is between bits 2 and 3
     \[ b_7 b_6 b_5 b_4 b_3 [.] b_2 b_1 b_0 \]
  2. The binary point is between bits 4 and 5
     \[ b_7 b_6 b_5 [.] b_4 b_3 b_2 b_1 b_0 \]

- 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 (e.g., 32) 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**
  - 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
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

❖ Floating point approximates real numbers
  ▪ We will discuss more details on Monday!
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 2\textsuperscript{nd} most significant byte of an int:

- First shift, then mask: \((x\gg16) \& 0xFF\)

<table>
<thead>
<tr>
<th></th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>(x\gg16)</td>
<td>00000000 00000000 00000001 00000010</td>
</tr>
<tr>
<td>0xFF</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td>((x\gg16) &amp; 0xFF)</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
</tbody>
</table>

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

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

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

0
1
0
1
Using Shifts and Masks

- Conditionals as Boolean expressions
  - For int \( x \), what does \((x<<31)>>31\) do?

<table>
<thead>
<tr>
<th></th>
<th>( x=!123 )</th>
<th>( x&lt;&lt;31 )</th>
<th>((x&lt;&lt;31)&gt;&gt;31)</th>
<th>(!x)</th>
<th>(!x&lt;&lt;31)</th>
<th>((!x&lt;&lt;31)&gt;&gt;31)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=!123</td>
<td>00000000 00000000 00000000 00000001</td>
<td>10000000 00000000 00000000 00000000</td>
<td>11111111 11111111 11111111 11111111</td>
<td>00000000 00000000 00000000 00000000</td>
<td>00000000 00000000 00000000 00000000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

- Can use in place of conditional:
  - In C: if \( x \) { \( a=y; \) } else { \( a=z; \) } equivalent to \( a=x?y:z; \)
  - \( a=(((x<<31)>>31)\&y)\ |\ (((!x<<31)>>31)\&z); \)