Integers II
CSE 351 Winter 2017

http://xkcd.com/571/
Administrivia

- No class Monday:

- After today, Lab 1 should be “easy” 😊
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C

- Consequences of finite width representations
  - Overflow, sign extension

- Shifting and arithmetic operations
Encoding Integers

- The hardware (and C) supports two flavors of integers
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives

- Cannot represent all integers with \( w \) bits
  - Only \( 2^w \) distinct bit patterns
  - Unsigned values: \( 0 \ldots 2^w - 1 \)
  - Signed values: \( -2^{w-1} \ldots 2^{w-1} - 1 \)

- **Example:** 8-bit integers (i.e., `char`)
Sign and Magnitude

- Designate the high-order bit (MSB) as the “sign bit”
  - \( \text{sign}=0 \): positive numbers; \( \text{sign}=1 \): negative numbers

- Positives:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0

- Examples (8 bits):
  - \( 0x00 = \underbrace{00000000}_2 \) is non-negative, because the sign bit is 0
  - \( 0x7F = \underbrace{01111111}_2 \) is non-negative \((+127_{10})\)
  - \( 0x85 = \underbrace{10000101}_2 \) is negative \((-5_{10})\)
  - \( 0x80 = \underbrace{10000000}_2 \) is negative... zero???
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude

- Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - Arithmetic is cumbersome
    - Example: $4 - 3 \neq 4 + (-3)$

- Negatives “increment” in wrong direction!
Two’s Complement

Let’s fix these problems:

1) “Flip” negative encodings so incrementing works
Two’s Complement

- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
  2) “Shift” negative numbers to eliminate −0

- MSB still indicates sign!

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Hex (Signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
<td>E</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Hex (Signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>1111</td>
<td>F</td>
<td>−1</td>
</tr>
<tr>
<td>−2</td>
<td>1110</td>
<td>E</td>
<td>−2</td>
</tr>
<tr>
<td>−3</td>
<td>1101</td>
<td>D</td>
<td>−3</td>
</tr>
<tr>
<td>−4</td>
<td>1100</td>
<td>C</td>
<td>−4</td>
</tr>
<tr>
<td>−5</td>
<td>1011</td>
<td>B</td>
<td>−5</td>
</tr>
<tr>
<td>−6</td>
<td>1010</td>
<td>A</td>
<td>−6</td>
</tr>
<tr>
<td>−7</td>
<td>1001</td>
<td>9</td>
<td>−7</td>
</tr>
<tr>
<td>−8</td>
<td>1000</td>
<td>8</td>
<td>−8</td>
</tr>
</tbody>
</table>

Diagram showing the mapping of decimal to binary and vice versa, with the distinction between positive and negative numbers.
Two’s Complement Negatives

- Accomplished with one neat mathematical trick!

\[ b_{w-1} \text{ has weight } -2^{w-1}, \text{ other bits have usual weights } +2^i \]

- **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 \(-1\)
Why Two’s Complement is So Great

- Roughly same number of (+) and (−) numbers
- Positive number encodings match unsigned
- Single zero
- All zeros encoding = 0

Simple negation procedure:
- Get negative representation of any integer by taking bitwise complement and then adding one!
  \( (~x + 1 == -x) \)
Unsigned vs. Two’s Complement

- 4-bit Example:

  Unsigned: \[1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]

  Two’s Complement: \[1 \times (-2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]

  (math) difference = 16 = \(2^4\)
Unsigned vs. Two’s Complement

- 4-bit Example:

Unsigned:
1011

Two’s Complement:
1x(-2^3) + 0x2^2 + 1x2^1 + 1x2^0

11

(math) difference = 16 = 2^4

-5

Unsigned

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

Two’s Complement

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
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:

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>-4</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>+011</td>
<td>+3</td>
<td>+011</td>
</tr>
<tr>
<td></td>
<td><strong>=7</strong></td>
<td><strong>=-1</strong></td>
<td><strong>=1</strong></td>
</tr>
<tr>
<td></td>
<td><strong>0111</strong></td>
<td><strong>1111</strong></td>
<td><strong>10001</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1100</th>
<th>-3</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>+011</td>
<td>+1101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td><strong>=0001</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>0111</strong></td>
<td><strong>=0001</strong></td>
<td></td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

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

  \[
  \begin{align*}
  \text{bit representation of } x & = \text{bit representation of } -x \\
  + & \quad 0 \\
  \end{align*}
  \]

  (ignoring the carry-out bit)

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

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

- For all representable positive integers \( x \), we want:

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

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

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

These are the bitwise complement plus 1!

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

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

- **Unsigned Values**
  - $\text{UMin} = 0b00...0 = 0$
  - $\text{UMax} = 0b11...1 = 2^w - 1$

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

- **Example:** Values for $w = 32$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>4,294,967,296</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>2,147,483,647</td>
<td>7F FF FF FF</td>
<td>01111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-2,147,483,648</td>
<td>80 00 00 00</td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
In C: Signed vs. Unsigned

- Casting
  - Bits are unchanged, just interpreted differently!
    - int tx, ty;
    - unsigned ux, uy;
  - Explicit casting
    - tx = (int) ux;
    - uy = (unsigned) ty;
  - Implicit casting can occur during assignments or function calls
    - tx = ux;
    - uy = ty;
    - gcc flag -Wsign-conversion produces warnings for implicit casts, but -Wall does not
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 <, >, ==, <=, >=
Casting Surprises

- 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>Op</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 0000 0000 0000 0000 0000 0000</td>
<td>==</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</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>Signed</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000</td>
<td>Signed</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>Unsigned</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 11110</td>
<td>Signed</td>
</tr>
<tr>
<td>(unsigned) -1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 11110</td>
<td>Unsigned</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>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>(int) 2147483648U 1000 0000 0000 0000 0000 0000 0000</td>
<td>Signed</td>
</tr>
</tbody>
</table>
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

- 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

- Computer handling of overflow
  - CPU *may be* capable of “throwing an exception” for overflow on signed values
  - CPU doesn’t throw exception for unsigned
  - C and Java ignore overflow exceptions... 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 \((-2^N)\)
  \[
  \begin{array}{c}
  15 \\
  + 2 \\
  \hline
  17 \\
  \text{Overflow}
  \end{array} \quad \begin{array}{c}
  1111 \\
  + 0010 \\
  \hline
  0001
  \end{array}
  \]

- **Subtraction**: borrow \((+2^N)\)
  \[
  \begin{array}{c}
  1 \\
  \text{-} \ 2 \\
  \hline
  -1 \\
  \text{Overflow}
  \end{array} \quad \begin{array}{c}
  10001 \\
  \text{-} \ 0010 \\
  \hline
  1111
  \end{array}
  \]

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

- **Addition:** \((+)+(+)=(-)\) result?
  
  \[
  \begin{array}{c}
  \hline
  \text{6} \\
  + \text{3} \\
  \hline
  \text{9} \\
  \hline
  \text{6} \\
  \end{array}
  \quad \begin{array}{c}
  \text{0110} \\
  + \text{0011} \\
  \hline
  \text{1001} \\
  \hline
  \text{7} \\
  \end{array}
  \]

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

  \[
  \begin{array}{c}
  \hline
  \text{7} \\
  - \text{3} \\
  \hline
  \text{6} \\
  \end{array}
  \quad \begin{array}{c}
  \text{1001} \\
  - \text{0011} \\
  \hline
  \text{0110} \\
  \hline
  \text{7} \\
  \end{array}
  \]

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
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**
    - 4-bit: 0010 = +2
    - 8-bit: 00000010 = +2
  - **Negative Case**
    - 4-bit: 1100 = -4
    - 8-bit: 00001100 = +12
    - Make MSB 1?
    - 4-bit: 1100 = -4
    - 8-bit: 10001100 = -116
    - Add 1’s?
    - 4-bit: 1100 = -4
    - 8-bit: 11111100 = -4
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
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0 \)
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>
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 0s 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 0s 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:**

- Shfts by \(n < 0\) or \(n \geq w\) (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 \(>>\)
Shifting Arithmetic?

- What are the following computing?
  - $x >> n$
    - $0b\ 0100 \gg 1 = 0b\ 0010$
    - $0b\ 0100 \gg 2 = 0b\ 0001$
    - **Divide by** $2^n$
  - $x << n$
    - $0b\ 0001 << 1 = 0b\ 0010$
    - $0b\ 0001 << 2 = 0b\ 0100$
    - **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: \( x \times 2^n \)?

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

- **Reminder:** C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values

- **Logical Shift:** $x/2^n$?

```plaintext
\[
x_u = 240u; \quad \text{11110000} = 240
\]
\[
R1_u = x_u \gg 3; \quad \text{00011110000} = 30
\]
\[
R2_u = x_u \gg 5; \quad \text{00000111110000} = 7
\]
```

- **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:** \( \frac{x}{2^n} \)?

\[
x_{\text{S}} = -16; \quad 11110000 = -16
\]
\[
R_{1\text{S}} = x_{\text{U}} >> 3; \quad 111111110000 = -2
\]
\[
R_{2\text{S}} = x_{\text{U}} >> 5; \quad 11111111110000 = -1
\]

*rounding (down)*
Peer Instruction Question

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

- 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)`
Using Shifts and Masks

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

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(x\gg 16)</th>
<th>0xFF</th>
<th>((x\gg 16) &amp; 0xFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>00000001 00000010 00000011 00000100</td>
<td>00000000 00000000 00000001 00000010</td>
<td>00000000 00000000 00000000 11111111</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 works in either case
    - Need mask to clear 1s possibly shifted in

<table>
<thead>
<tr>
<th></th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<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>

<table>
<thead>
<tr>
<th></th>
<th>10000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x &gt;&gt; 31</td>
<td>11111111 11111111 11111111 11111111</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 \( (x\ll31)\gg31 \) do?

<table>
<thead>
<tr>
<th>x=!!123</th>
<th>00000000 00000000 00000000 00000000 00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&lt;&lt;31</td>
<td>10000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(x&lt;&lt;31)\gg31</td>
<td>11111111 11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>!x</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>!x&lt;&lt;31</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(!x&lt;&lt;31)\gg31</td>
<td>00000000 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 \( a=x?y:z; \)
  - \( a=((x\ll31)\gg31)\&y) \mid (((!x\ll31)\gg31)\&z); \)
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpret* 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
  - Can be used in multiplication with constant or bit masking
  - Right shifting can be arithmetic (sign) or logical (0)