Integers II
CSE 351 Autumn 2020

Instructor: Justin Hsia
Teaching Assistants:
- Aman Mohammed
- Cosmo Wang
- Joy Dang
- Kyrie Dowling
- Yan Zhe Ong
- Ami Oka
- Hang Do
- Julia Wang
- Mariam Mayanja
- Callum Walker
- Jim Limprasert
- Kaelin Laundry
- Shawn Stanley

http://xkcd.com/571/
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:
  - 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

- 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?

\[ -2^5 = -32 \]

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

Signed, 2 bytes

- What is the result of the following expressions?

- (signed char) $uc \gg 2$

- (unsigned char) $uc \gg 3$

Signed: $0b\ 1010\ 0001 \xrightarrow{\text{arithmetic}} 0b\ 1110\ 1000 = 0x\ E8$

Unsigned: $0b\ 1010\ 0001 \xrightarrow{\text{logical}} 0b\ 0001\ 0100 = 0x\ 14$
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?

  \[
  \begin{align*}
  \text{00000001} &+ ????\\
  \hline
  \text{00000000}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{00000010} &+ ????\\
  \hline
  \text{00000000}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{11000011} &+ ????\\
  \hline
  \text{00000000}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{11000011} &+ ????\\
  \hline
  \text{00000000}
  \end{align*}
  \]
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?

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

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

\[ 2^{w-1} - 1 = 0b01\ldots1 \]

\[ -2^{w-1} = 0b10\ldots0 = T_{\text{Min}} \]

\[ U_{\text{Max}} = 0b1\ldots1 = 2^w - 1 \]

\[ 0b10\ldots0 = 2^{w-1} \]

\[ U_{\text{Max}} - 1 \]

\[ T_{\text{Max}} + 1 \]

\[ T_{\text{Max}} \]

\[ 0/U_{\text{Min}} \]
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>$U_{\text{Max}}$</td>
<td>18,446,744,073,709,551,615</td>
<td>FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>9,223,372,036,854,775,807</td>
<td>7F FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</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>
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

\[ 127 < \text{(signed char) 128u} \]

<table>
<thead>
<tr>
<th>Signed comparison:</th>
<th>Signed comparison:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01111111</td>
<td>0b10000000</td>
</tr>
<tr>
<td>\text{Signed}</td>
<td>\text{Signed}</td>
</tr>
<tr>
<td>\text{Comparison}</td>
<td>\text{Comparison}</td>
</tr>
<tr>
<td>127</td>
<td>-128</td>
</tr>
<tr>
<td>\text{False}</td>
<td>\text{False}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unsigned comparison:</th>
<th>Unsigned comparison:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01111111</td>
<td>0b10000000</td>
</tr>
<tr>
<td>127</td>
<td>128</td>
</tr>
<tr>
<td>\text{True}</td>
<td>(e.g., if LHS was 127u)</td>
</tr>
</tbody>
</table>
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 showing sign extension](image)
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 $\text{modulo } 2^w$
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 $U_{\text{Min}}$</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 $T_{\text{Max}}$</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8 $U_{\text{Max}}$</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9 $T_{\text{Min}}$</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10 $T_{\text{Min}}$</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 $U_{\text{Max}}$</td>
<td>-1</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

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

- **Subtraction**: borrow \(+2^N\)
  
  
  \[
  \begin{array}{c}
  1 \\
  - \ 2 \\
  \hline
  \underline{-1}
  \end{array}
  \quad
  \begin{array}{c}
  10001 \\
  - \ 0010 \\
  \hline
  1111
  \end{array}
  \quad
  \begin{array}{c}
  15
  \end{array}
  \]

\[±2^N \text{ because of modular arithmetic}\]

\[2^4 = 16\]
Overflow: Two’s Complement

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

- **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 the 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
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 \((x << n)\) bit vector \(x\) by \(n\) positions
  - Fill with 0’s on right
- Right shift \((x >> n)\) 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 \(x\))

8-bit example:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>0010 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
<td>(0010 0010)</td>
</tr>
<tr>
<td>(x &lt;&lt;= 3)</td>
<td></td>
<td>(0001 0000)</td>
</tr>
<tr>
<td>logical: (x &gt;&gt; 2)</td>
<td></td>
<td>(0000 1000)</td>
</tr>
<tr>
<td>arithmetic: (x &gt;&gt; 2)</td>
<td></td>
<td>(0000 1000)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>1010 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
<td>(1010 0010)</td>
</tr>
<tr>
<td>(x &lt;&lt;= 3)</td>
<td></td>
<td>(0001 0000)</td>
</tr>
<tr>
<td>logical: (x &gt;&gt; 2)</td>
<td></td>
<td>(0010 1000)</td>
</tr>
<tr>
<td>arithmetic: (x &gt;&gt; 2)</td>
<td></td>
<td>(1110 1000)</td>
</tr>
</tbody>
</table>
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!
    (compiler will try to optimize for you)

- **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: $x \times 2^n$?

<table>
<thead>
<tr>
<th>x = 25;</th>
<th>00011001 = 25 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1=x&lt;&lt;2; 001100100 = 100 100</td>
<td></td>
</tr>
<tr>
<td>L2=x&lt;&lt;3; 0011001000 = -56 200</td>
<td></td>
</tr>
<tr>
<td>L3=x&lt;&lt;4; 00110010000 = -112 144</td>
<td></td>
</tr>
</tbody>
</table>

- Signed overflow
- Unsigned overflow
Right Shifting Arithmetic 8-bit Examples

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

\[
x_{u} = 240u; \quad 11110000 = 240 \quad /8 = 30
\]

\[
R1_{u} = x_{u} >> 3; \quad 00011110000 \quad = 30 \quad /4 = 7.5
\]

\[
R2_{u} = x_{u} >> 5; \quad 0000011110000 \quad = 7 \quad \text{rounding (down)}
\]
Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Arithmetic Shift: $x / 2^n$?

$x_s = -16; \quad 11110000 = -16$

$R1_s = x_u >> 3; \quad 11111110000 = -2$

$R2_s = x_u >> 5; \quad 1111111110000 = -1$

(rounding (down))
Assume we are using 8-bit arithmetic:

- **x == (unsigned char) x**
  - Example: \( x = 0 \)
  - All solutions: works for all \( x \)

- **x >= 128U**
  - \( x = -1 \)
  - any \( x < 0 \)

- **x != (x>>2)<<2**
  - \( x = 3 \)
  - any \( x \) where lowest two bits are not 0b00

- **x == -x**
  - **Hint**: there are two solutions
  - \( x = 0 \)
  - ① \( x=0b0...0 = 0 \)
  - ② \( x=0b1...0 = -128 \)
  - any \( x \) where upper two bits are exactly 0b01

- **(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\(^{nd}\) most significant byte of an int
- Extract the sign bit of a signed int
- Conditionals as Boolean expressions
Using Shifts and Masks

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

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

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

<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 00000001</td>
</tr>
<tr>
<td>((x &gt;&gt; 31) &amp; 0x1)</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

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

| `x=!1123` | 00000000 00000000 00000000 00000000 00000001 |
| `x<<31` | 10000000 00000000 00000000 00000000 00000000 |
| `(x<<31)>>31` | 11111111 11111111 11111111 11111111 |
| `!x` | 00000000 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);`