Integers II
CSE 351 Autumn 2019

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu
Diya Joy
Maurice Montag
Suraj Jagadeesh

Antonio Castelli
Ivy Yu
Melissa Birchfield

Cosmo Wang
Kaelin Laundry
Millicent Li

http://xkcd.com/1953/
Administrivia

- hw4 due 10/7, hw5 due 10/9
- Lab 1a due Monday (10/7)
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b released today, due 10/14
  - Bit puzzles on number representation
  - Can start after today’s lecture, but floating point will be introduced next week
  - Section worksheet from yesterday has helpful examples, too
  - Bonus slides at the end of today’s lecture have relevant examples
Extra Credit

- All labs starting with Lab 1b have extra credit portions
  - These are meant to be fun extensions to the labs

- Extra credit points *don't* affect your lab grades
  - From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
  - Make sure you finish the rest of the lab before attempting any extra credit
Integers

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

- Consequences of finite width representations
  - Overflow, sign extension

- Shifting and arithmetic operations
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:**

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>+0011</td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>+0011</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>+1101</td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

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

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

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

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

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

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

- 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:
- 2’s Complement Range
- Unsigned Range
- Transition points:
  - TMin
  - 0
  - -1
  - -2
  - TMax
  - UMin
  - UMax
  - UMax - 1
  - TMax + 1
  - 0/UMin

Graphs:
- Two’s complement to unsigned
- unsigned to two’s complement
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 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 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 00 00</td>
</tr>
<tr>
<td>$-1$</td>
<td>-1</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>$0$</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 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 <, >, ==, <=, >=
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>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>0U</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>0U</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647</td>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>-2147483648</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647U</td>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>-2147483648</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>-2</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>-2</td>
<td>1111 1111 1111 1111 1111 1111 1111</td>
</tr>
<tr>
<td>2147483647</td>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>2147483648U</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647</td>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>(int) 2147483648U</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</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

- 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 \((-2^N)\)

\[
\begin{array}{c@{}c@{}c@{}c@{}c}
  1 & 5 & \rightarrow & 1 & 1 & 1 & 1 \\
+ & 2 & \rightarrow & 0 & 0 & 1 & 0 \\
\hline
  1 & 7 & \rightarrow & 1 & 0 & 0 & 0 & 1
\end{array}
\]

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

\[
\begin{array}{c@{}c@{}c@{}c@{}c}
  1 & \rightarrow & 1 & 0 & 0 & 0 & 1 \\
- & 2 & \rightarrow & 0 & 0 & 1 & 0 \\
\hline
  1 & 5 & \rightarrow & 1 & 1 & 1 & 1
\end{array}
\]

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

- **Addition:** $(+)+(+)=(-)$ result?
  
  \[
  \begin{array}{c}
  \phantom{+}6 \\
  + \phantom{0}3 \\
  \hline
  \phantom{+}9
  \end{array}
  \quad
  \begin{array}{c}
  0110 \\
  + 0011 \\
  \hline
  1001
  \end{array}
  \]
  
  \[\text{result: } -7\]

- **Subtraction:** $(-)+(-)=(+)$?
  
  \[
  \begin{array}{c}
  \phantom{-}7 \\
  - \phantom{0}3 \\
  \hline
  \phantom{-}4
  \end{array}
  \quad
  \begin{array}{c}
  1001 \\
  - 0011 \\
  \hline
  0110
  \end{array}
  \]
  
  \[\text{result: } 6\]

**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

Which of the following 8-bit numbers has the same signed value as the 4-bit number \(0b\underline{1100}\)?

- Underlined digit = MSB

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

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

```plaintext
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** \((x \ll n)\) bit vector \(x\) by \(n\) positions
  - Throw away (drop) extra bits on left
  - Fill with 0s on right

- **Right shift** \((x \gg n)\) 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:**
  - Shifts by \(n < 0\) or \(n \geq w\) (\(w\) is 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?
  - $x >> n$
    - $0b\ 0100\ >>\ 1\ =\ 0b\ 0010$
    - $0b\ 0100\ >>\ 2\ =\ 0b\ 0001$
    - **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: \( x \times 2^n \)?

<table>
<thead>
<tr>
<th></th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 25; )</td>
<td>00011001 = 25</td>
<td>00011001 = 25</td>
</tr>
<tr>
<td>( L1 = x &lt;&lt; 2; )</td>
<td>0001100100 = 100</td>
<td>0001100100 = 100</td>
</tr>
<tr>
<td>( L2 = x &lt;&lt; 3; )</td>
<td>00011001000 = -56</td>
<td>00011001000 = 200</td>
</tr>
<tr>
<td>( L3 = x &lt;&lt; 4; )</td>
<td>000110010000 = -112</td>
<td>000110010000 = 144</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: \( x / 2^n \)?

\[
x_u = 240u; \quad 11110000 = 240
\]

\[
R1u=xu>>3; \quad 00011110000 = 30
\]

\[
R2u=xu>>5; \quad 0000011110000 = 7
\]

rounding (down)
Right Shifting Arithmetic 8-bit Examples

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

\[
\begin{align*}
xs &= -16; \quad 11110000 = -16 \\
R1s &= xu >> 3; \quad 11111110000 = -2 \\
R2s &= xu >> 5; \quad 1111111110000 = -1
\end{align*}
\]
Practice Question

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

- Assume we are using 8-bit arithmetic:
  - \( x == \text{(unsigned char)} \ x \)
  - \( x >= 128U \)
  - \( x != (x>>2)\ll2 \)
  - \( x == -x \)
    - Hint: there are two solutions
  - \( (x < 128U) && (x > 0x3F) \)
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just interpreted differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
    - Type of variables affects behavior of operators (shifting, comparison)

- We can only represent so many numbers in \( w \) bits
  - When we exceed the limits, arithmetic overflow occurs
  - Sign extension tries to preserve value when expanding

- Shifting is a useful bitwise operator
  - Right shifting can be arithmetic (sign) or logical (0)
  - Can be used in multiplication with constant or bit masking
Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 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 \textit{byte} of an \texttt{int}:
  - First shift, then mask: $(x \gg 16) \& 0xFF$

\begin{tabular}{|c|c|}
\hline
\textbf{x} & 00000001 00000010 00000011 00000100 \\
\hline
\textbf{x}\gg16 & 00000000 00000000 00000001 00000010 \\
\hline
0xFF & 00000000 00000000 00000000 11111111 \\
\hline
(x\gg16) & 0xFF & 00000000 00000000 00000000 00000000 \\
\hline
\end{tabular}

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

\begin{tabular}{|c|c|}
\hline
\textbf{x} & 00000001 00000010 00000011 00000100 \\
\hline
0xFF0000 & 00000000 11111111 00000000 00000000 \\
\hline
\textbf{x} \& 0xFF0000 & 00000000 00000010 00000000 00000000 \\
\hline
(x\&0xFF0000)\gg16 & 00000000 00000000 00000000 000000010 \\
\hline
\end{tabular}
Using Shifts and Masks

- Extract the sign bit of a signed int:
  - First shift, then mask: \( x \gg 31 \) & 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 \gg 31 )</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>((x \gg 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 \gg 31 )</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>((x \gg 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?

<table>
<thead>
<tr>
<th></th>
<th>00000000 00000000 00000000 00000000 00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=!123</code></td>
<td>00000000 00000000 00000000 00000000 00000000 000000000001</td>
</tr>
<tr>
<td><code>x&lt;&lt;31</code></td>
<td>10000000 00000000 00000000 00000000 00000000 000000000000</td>
</tr>
<tr>
<td><code>(x&lt;&lt;31)&gt;&gt;31</code></td>
<td>11111111 11111111 11111111 11111111 11111111 1111111111 11111111</td>
</tr>
<tr>
<td><code>!x</code></td>
<td>00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>!x&lt;&lt;31</code></td>
<td>00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>(!!x&lt;&lt;31)&gt;&gt;31</code></td>
<td>00000000 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<<31)>>31)&y) | (((!x<<31)>>31)&z);`