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 \( \text{modulo} \ 2^w \)

- **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>+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:

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

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

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

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

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

- For all representable positive integers $x$, we want:
  
  \[
  x + (\overline{x} + 1) = \overline{0b1...1} \\
  x + (\overline{x}) = -1 \\
  x + (\overline{x} + 1) = 0 \\
  \overline{-x} = \overline{x} + 1
  \]

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

  \[
  \begin{array}{c}
  00000001 \\
  + 11111111 \\
  \hline
  100000000
  \\
  00000010 \\
  + 11111110 \\
  \hline
  100000000
  \\
  11000011 \\
  + 00111101 \\
  \hline
  100000000
  \end{array}
  \]

  These are the bitwise complement plus 1!

  \[
  -x = \overline{x} + 1
  \]
Signed/Unsigned Conversion Visualized

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

\[ 2^{w-1} - 1 = \text{Ob01...1} \]

\[ 2^{w-1} = \text{Ob10...0} = U\text{Min} \]
Values To Remember

- **Unsigned Values**
  - $U_{\text{Min}} = 0b00\ldots0 = 0$
  - $U_{\text{Max}} = 0b11\ldots1 = 2^w - 1$

- **Two’s Complement Values**
  - $T_{\text{Min}} = 0b10\ldots0 = -2^{w-1}$
  - $T_{\text{Max}} = 0b01\ldots1 = 2^{w-1} - 1$
  - $-1 = 0b11\ldots1$

- **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</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 FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>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;` (also implicitly occurs with printf format specifiers)
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* (unsigned “dominates”)
  - 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 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 1110</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 1110</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
- C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no warning/indication… oops!

### Table: Integer Encoding

<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</td>
<td>-8_{T\text{Min}}</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_{U\text{Max}}</td>
<td>-1</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

- **Addition**: drop carry bit ($-2^N$)
  
  \[
  \begin{array}{c}
  15 \\
  + \ 2 \\
  \hline
  17 \\
  \text{\color{red}1} \\
  \end{array}
  \quad
  \begin{array}{c}
  1111 \\
  + \ 0010 \\
  \hline
  \text{\color{red}1}0001 \\
  \end{array}
  \]

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

$\pm 2^N$ 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 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 \( \text{0b1100} \)?

- Underlined digit = MSB

A. 0b 0000 1100 (add zeros)
B. 0b 1000 1100 (add leading 1)
C. 0b 1111 1100 (add ones)
D. 0b 1100 1100 (duplicate)
E. We’re lost...

\[
\begin{align*}
\text{Underline digit = MSB: } & \quad 0100 \\
\text{Add 0s: } & \quad 0000 1100 \\
\text{Add 1: } & \quad 1000 1100 \\
\text{Add 1s: } & \quad 1111 1100 \\
\text{Duplicate: } & \quad 1100 1100 \\
\end{align*}
\]
Sign Extension

- **Task:** Given a $w$-bit signed integer $X$, convert it to a $w+k$-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.
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-2}, ..., x_1, x_0$
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** ($x << n$) 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:**
- Shifts by \(n < 0\) or \(n \geq w\) (bit width of \(x\)) are **undefined**: behavior not guaranteed
- **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?
  - \( x >>> n \)
    - \( \text{0b 0100} \ggg 1 = \text{0b 0010} \)
    - \( \text{0b 0100} \ggg 2 = \text{0b 0001} \)
    - \text{Divide by } 2^n
  - \( x <<= n \)
    - \( \text{0b 0001} \lll 1 = \text{0b 0010} \)
    - \( \text{0b 0001} \lll 2 = \text{0b 0100} \)
    - \text{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: \( x \times 2^n \)?

\[
\begin{align*}
\text{x} &= 25; \quad 00011001 = 25 \quad 25 \\
\text{L1} &= \text{x} \ll 2; \quad 0001100100 = 100 \quad 100 \\
\text{L2} &= \text{x} \ll 3; \quad 00011001000 = -56 \quad 200 \\
\text{L3} &= \text{x} \ll 4; \quad 00110010000 = -112 \quad 144
\end{align*}
\]

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

$xu = 240u; \quad 11110000 = 240$

$R1u=xu>>3; \quad 00011110000 \quad = \quad 30$ \quad ($\frac{1}{8} = 30$)

$R2u=xu>>5; \quad 0000011110000 \quad = \quad 7$ \quad ($\frac{1}{4} = 7.5$)

(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: $x / 2^n$?

For $xs = -16$:

- $11110000 = -16$
- $R1s = xu >> 3; 11111110000 = -2_{\text{rounding (down)}}$
- $R2s = xu >> 5; 1111111110000 = -1$

Note: $\frac{1}{4} = -0.5$
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:
  - \( x == (\text{unsigned char}) x \)
  - \( x >= 128U \)
  - \( x != (x>>2) << 2 \)
  - \( x == -x \)
    - 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: \((x\gg\!16) \& \texttt{0xFF}\)
    
    \[
    \begin{array}{c|c}
    x & 00000001 00000010 00000011 00000100 \\
    \hline
    x\gg16 & 00000000 00000000 00000001 00000010 \\
    \hline
    \texttt{0xFF} & 00000000 00000000 00000000 11111111 \\
    \hline
    (x\gg16) \& \texttt{0xFF} & 00000000 00000000 00000000 00000010
    \end{array}
    \]

  - Or first mask, then shift: \((x \& \texttt{0xFFFF0000}) \gg16\)
    
    \[
    \begin{array}{c|c}
    x & 00000001 00000010 00000011 00000100 \\
    \hline
    \texttt{0xFFFF0000} & 00000000 11111111 00000000 00000000 \\
    \hline
    x \& \texttt{0xFFFF0000} & 00000000 00000010 00000000 00000000 \\
    \hline
    (x\&\texttt{0xFFFF0000}) \gg16 & 00000000 00000000 00000000 00000010
    \end{array}
    \]
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?

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

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