Integers II
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee Natalie Andreeva Teagan Horkan

http://xkcd.com/1953/
Administrivia

- Lab 1a due Friday (6/29)
- Lab 1b due next Thursday (7/5)
  - Bonus slides at the end of today’s lecture have relevant examples
- Homework 2 released today, due two Wed from now (7/11)
  - Can start on Integers, will need to wait for Assembly
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\textsuperscript{\text{Umin}}</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\textsuperscript{\text{TMax}}</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8\textsuperscript{\text{TMin}}</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\textsuperscript{\text{UMax}}</td>
<td>-1</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

- **Addition:** drop carry bit ($-2^N$)

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

  \[
  1
  \]

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

  \[
  \begin{array}{c}
  1 \\
  - 2 \\
  \hline
  -1
  \end{array}
  \quad
  \begin{array}{c}
  10001 \\
  - 0010 \\
  \hline
  1111
  \end{array}
  \]

  \[
  15
  \]

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

- **Addition:** \((+ +) = (-)\) result?

\[
\begin{array}{c}
6 \\
+ 3 \\
\hline
9 \\
\hline
\textcolor{red}{-7}
\end{array}
\quad \begin{array}{c}
0110 \\
+ 0011 \\
\hline
1001
\end{array}

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

\[
\begin{array}{c}
-7 \\
- 3 \\
\hline
-10 \\
\hline
6
\end{array}
\quad \begin{array}{c}
1001 \\
- 0011 \\
\hline
0110
\end{array}
\]

**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?
Peer Instruction 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. \(0b\ 0000\ 1100\) (add zeros)
B. \(0b\ 1000\ 1100\) (add leading 1)
C. \(0b\ \underline{1111}\ 1100\) (add ones)
D. \(0b\ \underline{1100}\ 1100\) (duplicate)
E. We’re lost...

\[
\begin{align*}
\text{Underline digit = MSB} & \quad \text{positive!} \\
\text{Vote at [http://PollEv.com/justinh](http://PollEv.com/justinh)} & \\
\text{A. } 0b\ 0000\ 1100 \quad \text{(add zeros)} & \quad \text{much too negative: } -2^3 + 2^1 + 2^0 = -116 \\
\text{B. } 0b\ 1000\ 1100 \quad \text{(add leading 1)} & \quad \text{correct! } -y = 0b\ 0000\ 0011 + 1 = 11, \quad y = -4 \\
\text{C. } 0b\ \underline{1111}\ 1100 \quad \text{(add ones)} & \quad -2^3 + 2^2 + 2^1 + 2^0 = -52 \\
\text{D. } 0b\ \underline{1100}\ 1100 \quad \text{(duplicate)} & \quad \text{We’re lost...}
\end{align*}
\]
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
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_1, x_0$
Sign Extension Example

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

```
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?
  - $x >> n$
    - $0b\ 0100 >> 1 = 0b\ 0010$
    - $0b\ 0100 >> 2 = 0b\ 0001$
    - 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 Shift Arithmetic 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: \( x \times 2^n \)?

\[
\begin{align*}
x &= 25; & 00011001 &= 25 & 25 \\
L1 &= x \ll 2; & 001100100 &= 100 & 100 \\
L2 &= x \ll 3; & 0011001000 &= -56 & 200 \\
L3 &= x \ll 4; & 10010000 &= -112 & 144
\end{align*}
\]

Signed overflow

Unsigned overflow
Right Shift Arithmetic 8-bit Example

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

\[
\begin{align*}
xu &= 240u; \quad 11110000 = 240 \quad \frac{1}{8} = 30 \\
R1u &= xu >> 3; \quad 00011110000 = 30 \quad \frac{1}{4} = 7.5 \\
R2u &= xu >> 5; \quad 0000011110000 = 7 \quad \text{rounding (down)}
\end{align*}
\]
Right Shift Arithmetic 8-bit Example

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

```
x = -16; 11110000 = -16
R1s=xu>>3; 11111110000 = -2
R2s=xu>>5; 1111111110000 = -1
```

- Rounding (down)
Peer Instruction 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:
  - \( x \equiv (\text{unsigned char}) \times \)
  - \( x \geq 128 \text{U} \)
  - \( x \neq (x>>2)\ll 2 \)
  - \( x = -x \)
    - Hint: there are two solutions
  - \( (x < 128 \text{U}) \&\& (x > 0\times3F) \)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Example:</th>
<th>General:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \equiv (\text{unsigned char}) \times )</td>
<td>( x=0 )</td>
<td>all ( x ) works</td>
</tr>
<tr>
<td>( x \geq 128 \text{U} )</td>
<td>( x=-1 )</td>
<td>any ( x &lt; 0 )</td>
</tr>
<tr>
<td>( x \neq (x&gt;&gt;2)\ll 2 )</td>
<td>( x=3 )</td>
<td>any ( x ) where lowest two bits are not 0b00</td>
</tr>
<tr>
<td>( x = -x )</td>
<td>( x=0 )</td>
<td>0 x=0b0...0=0 ( x=0b10...0=-128 )</td>
</tr>
<tr>
<td>( (x &lt; 128 \text{U}) &amp;&amp; (x &gt; 0\times3F) )</td>
<td>( x=-64 )</td>
<td>any ( x ) where upper two bits are exactly 0b01</td>
</tr>
</tbody>
</table>
Unsigned Multiplication in C

Operands:

\(w\) bits

\[ u \]

\[ \ast \]

\[ v \]

True Product:

\(2w\) bits

\[ u \cdot v \]

Discard \(w\) bits:

\(w\) bits

\[ \text{UMult}_w(u, v) \]

- Standard Multiplication Function
  - Ignores high order \(w\) bits
- Implements Modular Arithmetic
  - \(\text{UMult}_w(u, v) = u \cdot v \mod 2^w\)
Multiplication with shift and add

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

**Operands: $w$ bits**

```
  u
  k

  * 2^k
```  

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

```
  u * 2^k
```

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

```
  UMult_w(u, 2^k)
  TMult_w(u, 2^k)
```

- **Examples:**
  - $u << 3 = u * 8$
  - $u << 5 - u << 3 = u * 24 \rightarrow 24 = 32 - 8$
  - $u << 4 + u << 3 = u * 24 \rightarrow 24 = 16 + 8$
  - Most machines shift and add faster than multiply
    - *Compiler generates this code automatically*
Number Representation Revisited

- What can we represent in one word?
  - 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:

  \[ xx.yyyy \]

  \[
  \begin{align*}
  2^1 & \quad 2^0 & \quad 2^{-1} & \quad 2^{-2} & \quad 2^{-3} & \quad 2^{-4} \\
  & & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} \\
  & & 0.5 & 0.25 & 0.125 & 0.0625
  \end{align*}
  \]

- **Example:** \( 10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10} \)

- Binary point numbers that match the 6-bit format above range from 0 \((00.0000_2)\) to 3.9375 \((11.1111_2)\)

  \[
  +00.0001_2 = 4_{10}
  \]

  \[
  100.0000_2 = 4 - 2^{-4}
  \]
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: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Example: $6.02_{10} \times 10^{23}$

- Mantissa
- Exponent
- Decimal point
- Radix (base)
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
  - Shift 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 exponent 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

\[
2^3 + 2^4 + 2^0 + 2^{-2} + 2^{-3} = 1011.011_2 = 1.011011 \times 2^3
\]

### Practice:
- Convert \(\frac{1}{5}\) to binary

\[
\frac{1}{5} - \frac{1}{8} = \frac{3}{40}, \quad \frac{3}{40} - \frac{1}{16} = \frac{1}{80} = \frac{1}{16} \left( \frac{1}{5} \right)
\]

\[
\frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \left( \frac{1}{5} \right)
\]

\[0.0011\]
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. We will try to cover these in lecture or section if we have the time.

- Extract the 2nd 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: \((x \gg 16) \& 0xFF\)
    - \(x\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
    - \(x \gg 16\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
    - \(0xFF\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      1 & 1 & 1 & 1 \\
      \end{array}
      \]
    - \((x \gg 16) \& 0xFF\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
  - Or first mask, then shift: \((x \ & \ 0xFF0000) >> 16\)
    - \(x\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
    - \(0xFF0000\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      1 & 1 & 1 & 1 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
    - \(x \ & \ 0xFF0000\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
    - \((x \& 0xFF0000) >> 16\):
      
      \[
      \begin{array}{cccc}
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      0 & 0 & 0 & 0 \\
      \end{array}
      \]
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: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z;`
  - `a=((x<<31)>>31)&y) | (((!x<<31)>>31)&z);`