Integers II
CSE 351 Autumn 2017

Instructor: Justin Hsia
Teaching Assistants: Lucas Wotton, Michael Zhang, Parker DeWilde, Ryan Wong, Sam Gehman, Sam Wolfson, Savanna Ye, Vinny Palaniappan

[Image: https://xkcd.com/557/]

Administivia
- Lab 1 due next Friday (10/13)
- Prelim submission (3+ of bits.c) due on Monday (10/9)
- Bonus slides at the end of today’s lecture have relevant examples

Integers
- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations

Peer Instruction Question
- Take the 4-bit number encoding \( x = 0b1011 \)
- Which of the following numbers is NOT a valid interpretation of \( x \) using any of the number representation schemes discussed last lecture?
  - A. -4
  - B. -5
  - C. 11
  - D. -3
  - E. We’re lost...

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^n
- 4-bit Examples:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
</tr>
<tr>
<td>-3</td>
<td>+1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=7</td>
<td>=-1</td>
<td>=1</td>
<td></td>
</tr>
</tbody>
</table>

Why Does Two’s Complement Work?
- For all representable positive integers \( x \), we want:
  \[
  \text{bit representation of } x + \text{bit representation of } -x \quad \text{[ignoring the carry-out bit]}
  \]
- What are the 8-bit negative encodings for the following?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>00000010</td>
<td>11000111</td>
</tr>
<tr>
<td>+????????</td>
<td>+????????</td>
<td>+????????</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

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

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

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

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

These are the bitwise complement plus 1!

\[-x = \sim x + 1\]

Signed/Unsigned Conversion Visualized

- Two’s Complement \( \rightarrow \) Unsigned
  - Ordering Inversion
  - Negative \( \rightarrow \) Big Positive

Values To Remember

- Unsigned Values
  - \( U_{\text{Min}} = 0b00\ldots 0 \)
  - \( U_{\text{Max}} = 0b11\ldots 1 \)

- Two’s Complement Values
  - \( T_{\text{Min}} = 0b10\ldots 0 \)
  - \( T_{\text{Max}} = 0b01\ldots 1 \)
  - \( -1 = 0b11\ldots 1 \)

Example: Values for \( w = 64 \)

<table>
<thead>
<tr>
<th>( U_{\text{Max}} )</th>
<th>( U_{\text{Max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>38,446,744,071,709,551,615</td>
<td></td>
</tr>
<tr>
<td>9,223,372,036,854,775,807</td>
<td></td>
</tr>
<tr>
<td>-9,223,372,036,854,775,808</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

In C: Signed vs. Unsigned

- Casting
  - Bits are unchanged, just interpreted differently!
    - \( \text{int } tx, ty; \)
    - \( \text{unsigned int } ux, uy; \)
  - Explicit casting
    - \( tx = (\text{int}) ux; \)
    - \( uy = (\text{unsigned int}) ty; \)
  - 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 \(<, >, ==, <=, >=\)
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
  - Shifting and arithmetic operations

---

Overflow: Unsigned

- Addition: drop carry bit (\(-2^N\))
  - 15
  - + 2
  - 0010
  - 1
- Subtraction: borrow (+2^N)
  - 1
  - - 2
  - 1001
  - - 0010
  - 1111

\[+2^N\] because of modular arithmetic

---

Overflow: Two's Complement

- Addition: (+) + (+) = (--) result?
  - 6
  - + 3
  - 0011
  - - 7
- Subtraction: (--)) + (-) = (+)?
  - 7
  - - 3
  - 0011
  - + 6
  - 0110

---

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

---

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!

---

Peer Instruction Question

- Which of the following 8-bit numbers has the same signed value as the 4-bit number 0b1100?
  - Underlined digit = MSB
  - Vote at http://PollEv.com/justinh

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
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_1, x_0 \)

[Diagram]

Sign Extension Example

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

<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</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>x</td>
<td>12345</td>
<td>00</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>FF</td>
<td>11001111 11001111</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>FF</td>
<td>11001111 11001111</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 \( x<<n \):**
  - Fill with \( 0 \)s on right
- **Right shift \( x>>n \):**
  - Logical shift (for unsigned values)
    - Fill with \( 0 \)s 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
  - 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?**
  - \( x>>n \)
    - \( 0b \ 0010 >> 1 = 0b \ 0010 \)
    - \( 0b \ 0010 >> 2 = 0b \ 0001 \)
    - **Divide by \( 2^n \)**
  - \( x<<n \)
    - \( 0b \ 0001 << 1 = 0b \ 0010 \)
    - \( 0b \ 0001 << 2 = 0b \ 0001 \)
    - **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: $x \times 2^n$?
  - Signed: $x = 25$; 00011001
  - Signed: $L1 = x << 2 = 25$ 25
  - Signed: $L2 = x << 3 = 100$ 100
  - Signed: $L3 = x << 4 = -56$ 200
  - Signed: $L3 = x << 4 = -112$ 144

Right Shifting Arithmetic 8-bit Examples

- Reminder: C operator $>>$ does logical shift on unsigned values and arithmetic shift on signed values
  - Logical Shift: $x / 2^n$?
  - Logical: $xu = 240u$; 11110000
  - Logical: $R1u = xu >> 3 = 30$
  - Logical: $R2u = xu >> 5 = 7$

Peer Instruction Question

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

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

Bonus Slides

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: \((x\ll16) \& 0xFF\)
  - Or first mask, then shift: \((x \& 0xFF0000) \gg 16\)

Using Shifts and Masks

- Extract the sign bit of a signed int:
  - First shift, then mask: \((x\ll31) \& 0x1\)
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear 1s possibly shifted in

Using Shifts and Masks

- Conditionals as Boolean expressions
  - For int \(x\), what does \((x\ll31)\gg31\) do?
    - Can use in place of conditional:
      - In C, if \(x\) \(a=y\) \(\text{else} \ (a=z)\) equivalent to \(a=x?y:z\);
      - \(a=((x\ll31)\gg31)&y) \| ((!(x\ll31))\gg31)\&z\);