Integers II
CSE 351 Winter 2018

Instructor:
Mark Wyse
Teaching Assistants:
Kevin Bi Parker, DevWilde, Emily Furst, Sarah House, Waylon Huang, Vinny Palaniappan

Administrivia
❖
Lab 1 due next Friday (1/19)
▪ Prelim submission (3+ of bits.c) due on Monday (1/15)
▪ Bonus slides at the end of today’s lecture have relevant examples
❖
HW 2 will be released on Monday 1/15
▪ Due Wednesday, 1/24
❖
No class on Monday 1/15

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 Negatives
❖ Accompilshed with one neat mathematical trick!

- $b_{w-1}$ has weight $-2^{w-1}$, other bits have usual weights $2^i$.

- 4-bit Examples:
  - 1010, unsigned: $1\cdot2^3+0\cdot2^2+1\cdot2^1+0\cdot2^0 = 10$
  - 1010, two’s complement: $-1\cdot2^3+0\cdot2^2+1\cdot2^1+0\cdot2^0 = -6$

- 1 represented as:
  - $1111 = 2^4(2^3-1)$
  - MSB makes it super negative, add up all the other bits to get back up to -1

Why Does Two’s Complement Work?
❖ For all representable positive integers $x$, we want:
  - bit representation of $x$
  - bit representation of $-x$ (ignoring the carry-out bit)

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

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 $2^w$

- 4-bit Example:

  |  4  | 1100 |  0100 |
  | +3  | +0011 | -3   | +1101 |
  | =1  | =1  | =1   |
Why Does Two’s Complement Work?

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

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

<table>
<thead>
<tr>
<th>8-bit representation of (-x)</th>
<th>8-bit representation of (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>11111110</td>
</tr>
<tr>
<td>11111111</td>
<td>00000000</td>
</tr>
<tr>
<td>00000010</td>
<td>11111110</td>
</tr>
<tr>
<td>11111111</td>
<td>00000000</td>
</tr>
<tr>
<td>00000010</td>
<td>11111110</td>
</tr>
<tr>
<td>11111111</td>
<td>00000000</td>
</tr>
<tr>
<td>00000010</td>
<td>11111110</td>
</tr>
<tr>
<td>11111111</td>
<td>00000000</td>
</tr>
</tbody>
</table>

These are the bitwise complement plus 1!

\(-x = \bar{x} + 1\)

Signed/Unsigned Conversion Visualized

❖ Two’s Complement → Unsigned

▪ Ordering Inversion

▪ Negative → Big Positive

Values To Remember

❖ Unsigned Values

- \( U_{\text{Min}} = \) \(0b00\ldots0\) = 0
- \( U_{\text{Max}} = \) \(0b11\ldots1\) = \(2^{w-1}\)

❖ Two’s Complement Values

- \( T_{\text{Min}} = \) \(0b10\ldots0\) = \(-2^{w-1}\)
- \( T_{\text{Max}} = \) \(0b01\ldots1\) = \(2^{w-1} - 1\)

Decimal | Hex
--- | ---
\( U_{\text{Max}} \) | \( \text{FF FF FF FF FF FF FF FF} \)
\( T_{\text{Min}} \) | \( -\text{FF FF FF FF FF FF FF FF} \)
\( U_{\text{Max}} \) | \( 18,446,744,073,709,551,615 \)
\( T_{\text{Max}} \) | \( 9,223,372,036,854,775,807 \)
\( T_{\text{Min}} \) | \( -9,223,372,036,854,775,808 \)
\( 0 \) | \( 0 \)

In C: Signed vs. Unsigned

❖ Casting

- Bits are unchanged, just interpreted differently!
- int \( x, y \);
- unsigned int \( x, y \);
- Explicit casting
  - \( x = \) (int) \( y \);
- \( y = \) (unsigned int) \( x \);
- Implicit casting can occur during assignments or function calls
  - \( x = y \);
  - \( y = x \);

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 \( \leq, \geq \Rightarrow \text{unsigned} \)

Casting Surprises

❖ 32-bit examples:

- \( T_{\text{Min}} - 2,147,483,648 \), \( T_{\text{Max}} - 2,147,483,647 \)

- \( T_{\text{Min}} \) - \( 0 \times 2^{31} \) - \( 0 \times 2^{30} \) - \( 0 \times 2^{29} \) - \( 0 \times 2^{28} \) - \( 0 \times 2^{27} \) - \( 0 \times 2^{26} \) - \( 0 \times 2^{25} \) - \( 0 \times 2^{24} \) - \( 0 \times 2^{23} \) - \( 0 \times 2^{22} \) - \( 0 \times 2^{21} \) - \( 0 \times 2^{20} \) - \( 0 \times 2^{19} \) - \( 0 \times 2^{18} \) - \( 0 \times 2^{17} \) - \( 0 \times 2^{16} \) - \( 0 \times 2^{15} \) - \( 0 \times 2^{14} \) - \( 0 \times 2^{13} \) - \( 0 \times 2^{12} \) - \( 0 \times 2^{11} \) - \( 0 \times 2^{10} \) - \( 0 \times 2^{9} \) - \( 0 \times 2^{8} \) - \( 0 \times 2^{7} \) - \( 0 \times 2^{6} \) - \( 0 \times 2^{5} \) - \( 0 \times 2^{4} \) - \( 0 \times 2^{3} \) - \( 0 \times 2^{2} \) - \( 0 \times 2^{1} \) - \( 0 \times 2^{0} \)
- \( U_{\text{Max}} \) - \( 9,223,372,036,854,775,807 \)
- \( T_{\text{Max}} \) - \( 2,147,483,647 \)
- \( T_{\text{Min}} \) - \( -2,147,483,648 \)
- \( U_{\text{Min}} \) - \( 0 \)
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!

Overflow: Two’s Complement
- Addition: \((+\cdot + = -)\) result?
  - \(6 + 3 = 1010\)
  - \(-7 + 7 = 0\)
- Subtraction: \((-\cdot - = +)\) result?
  - \(-7 - 3 = 0011\)
  - \(6 - 6 = 1010\)

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
    - Sign bit is 1
    - 4-bit: \(1010 = -2\)
    - 8-bit: \(11000010 = -2\)
    - Original: \(00000010\)

Overflow: Unsigned
- Addition: drop carry bit \((-2^N)\)
- Subtraction: borrow \((+2^N)\)

Overflow: Unsigned
- Addition: drop carry bit \((-2^N)\)
- Subtraction: borrow \((+2^N)\)

Overflow: Unsigned
- Addition: drop carry bit \((-2^N)\)
- Subtraction: borrow \((+2^N)\)
Peer Instruction Question
❖ Which of the following 8-bit numbers has the same signed value as the 4-bit number 0b1100?
   ▪ 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 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>00 39</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>ia</td>
<td>12345</td>
<td>00 00 39</td>
<td>00000000 00111001</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>FF FF</td>
<td>11111111 11000000</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF</td>
<td>11111111 11000000</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

Shifting Arithmetic?
❖ What are the following computing?
  ▪ x>>n
    - 0b 0100 >> 1 = 0b 0010
    - 0b 0100 >> 2 = 0b 0001
  ▪ Divide by 2^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: \( x \times 2^n \)

\[
\begin{align*}
  x &= 25; \quad 00011001 = 25 \quad 25 \\
  L1 &= x \ll 2; \quad 00011000100 = 100 \quad 100 \\
  L2 &= x \ll 3; \quad 00011001000 = -56 \quad 200 \\
  L3 &= x \ll 4; \quad 000110010000 = -112 \quad 144
\end{align*}
\]

Right Shifting Arithmetic 8-bit Examples

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

\[
\begin{align*}
  xu &= 240u; \quad 11110000 = 240 \\
  R1u &= xu >> 3; \quad 00011110000 = 30 \\
  R2u &= xu >> 5; \quad 0000011110000 = 7
\end{align*}
\]

- Arithmetic Shift: \( x/2^n \)

\[
\begin{align*}
  xs &= -16; \quad 11110000 = -16 \\
  R1s &= xu >> 3; \quad 11111110000 = -2 \\
  R2s &= xu >> 5; \quad 111111110000 = -1
\end{align*}
\]

Peer Instruction Question

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

- Assume we are using 8-bit arithmetic:
  - \( x \equiv (\text{unsigned char}) x \)
  - \( x >>= 128U \)
  - \( x != (x>>2)<<2 \)
  - \( x == -x \)
  - 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 2\text{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: \((x & 0xFF0000) >> 16\)

\[
\begin{array}{c|cccc}
\hline
x & 0000001 & 0000010 & 0000011 & 0000100 \\
\hline
x >> 16 & 0000000 & 0000000 & 0000000 & 0000001 \\
0xFF & 0000000 & 0000000 & 0000000 & 0000001 \\
(x >> 16) \& 0xFF & 0000000 & 0000000 & 0000000 & 0000001 \\
\hline
\end{array}
\]

❖ 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

\[
\begin{array}{c|cccc}
\hline
x & 00000000000000000000000000001001 \\
\hline
x >> 31 & 00000000000000000000000000000001 \\
0x1 & 00000000000000000000000000000001 \\
(x >> 31) \& 0x1 & 00000000000000000000000000000001 \\
\hline
\end{array}
\]

❖ Conditionals as Boolean expressions
▪ For int \(x\), what does \((x << 31) >> 31\) do?

\[
\begin{array}{c|c}
\hline
x & \text{condition} \\
\hline
!!123 & 00000000000000000000000000000001 \\
x << 31 & 00000000000000000000000000000000 \\
(x << 31) >> 31 & 00000000000000000000000000000000 \\
!x & 00000000000000000000000000000001 \\
!(x << 31) >> 31 & 00000000000000000000000000000001 \\
\hline
\end{array}
\]

▪ Can use in place of conditional:
  • in C: \(\text{if}(x) \{a=y;\} \text{else} \{a=z;\}\) equivalent to \(a=x ? y : z\)
  • \(a=(x<<(c<<31))\&y) \mid ((!a<<(c<<31))\&z)\)