Integers II
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/571/
Administrivia

- Lab 1 due next Friday (10/14)
  - Partial due on Monday (10/10)
- Homework 1 will be released on Tuesday (10/11)
  - Timing overlaps a bit with Lab 1
- No Panopto for this course
  - Online lecture videos from 2013 at:
Integers

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

- Consequences of finite width representations
  - Overflow, sign extension

- Shifting and arithmetic operations
Unsigned vs. Two’s Complement

- 4-bit Example:
  - Unsigned: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
  - Two’s Complement: $1 \times (-2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

- (math) difference = 16 = $2^4$

- Two’s Complement
  - $0 \rightarrow +1$
  - $1 \rightarrow +2$
  - $-2 \rightarrow +3$
  - $-3 \rightarrow +4$
  - $-4 \rightarrow +5$
  - $-5 \rightarrow +6$

- Unsigned
  - $0 \rightarrow +1$
  - $1 \rightarrow +2$
  - $2 \rightarrow +3$
  - $3 \rightarrow +4$
  - $4 \rightarrow +5$
  - $5 \rightarrow +6$

- 4-bit Example:
  - 1011

- 1011

- 11

- 11

- -5
Unsigned vs. Two’s Complement

- 4-bit Example:

Unsigned: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

Two’s Complement: $1 \times (-2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

(math) difference = $16 = 2^4$
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></th>
<th>0100</th>
<th>1100</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td>0011</td>
<td>1101</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td>+0011</td>
<td>+1101</td>
</tr>
<tr>
<td>=7</td>
<td>=0111</td>
<td>=1111</td>
<td>=1</td>
</tr>
<tr>
<td></td>
<td>=0101</td>
<td>=10011</td>
<td>=0001</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:
  
  \[
  \text{bit representation of } x + \text{bit representation of } -x = \text{bit representation of } -x
  \]

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

  \[
  \begin{array}{c}
  00000001 + \text{????????} = \text{????????} \\
  00000010 + \text{????????} = \text{????????} \\
  11000011 + \text{????????} = \text{????????}
  \end{array}
  \]
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:
  \[
  \text{bit representation of } x + \text{bit representation of } -x \equiv 0
  \]
  (ignoring the carry-out bit)

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

  \[
  \begin{align*}
  00000001 & \quad 00000010 & \quad 11000011 \\
  + 11111111 & + 11111110 & + 00111101 \\
  100000000 & + 100000000 & + 100000000
  \end{align*}
  \]

  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:
- Signed Range
  - 2’s Complement
  - TMin: -2
  - Tmax: 0
- Unsigned Range
  - UMax: 9
  - UMax - 1
  - Tmax + 1
  - Order inversion

Legend:
- Signed
- Unsigned
- Signed/Unsigned Conversion
- Ordering Inversion
- Negative → Big Positive
Values To Remember

- **Unsigned Values**
  - \( \text{UMin} = 0b00...0 = 0 \)
  - \( \text{UMax} = 0b11...1 = 2^w - 1 \)

- **Two’s Complement Values**
  - \( \text{TMin} = 0b10...0 = -2^{w-1} \)
  - \( \text{TMax} = 0b01...1 = 2^{w-1} - 1 \)
  - \(-1 = 0b11...1\)

- **Example: Values for \( w = 32 \)**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>FF FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>7F FF FF FF</td>
<td>01111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>80 00 00 00</td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>00 00 00 00</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
In C: Signed vs. Unsigned

- **Casting**
  - Bits are unchanged, just interpreted differently!
    - int tx, ty;
    - unsigned ux, uy;
  - *Explicit* casting
    - tx = (int) ux;
    - uy = (unsigned) ty;
  - *Implicit* casting can occur during assignments or function calls
    - tx = ux;
    - uy = ty;
    - gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not
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>Op</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0U</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>Signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0U</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>Signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>-2</td>
<td>Signed</td>
</tr>
<tr>
<td><em>(unsigned)</em> -1</td>
<td>&gt;</td>
<td>-2</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>*(int) 2147483648U</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

- Computer handling of overflow
  - CPU *may be* capable of “throwing an exception” for overflow on *signed* values
  - CPU doesn’t throw exception for *unsigned*
  - C and Java ignore overflow exceptions... oops!
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}
\quad
1

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

\[
\begin{array}{c}
1 \\
- 2 \\
\hline
-1
\end{array}
\quad
\begin{array}{c}
10001 \\
- 0010 \\
\hline
1111
\end{array}
\quad
15
\]

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

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

\[
\begin{array}{ccc}
6 & 0110 \\
+ 3 & + 0011 \\
\hline
9 & 1001 \\
\end{array}
\]

\(-7\)

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

\[
\begin{array}{ccc}
-7 & 1001 \\
- 3 & - 0011 \\
\hline
-10 & 0110 \\
\end{array}
\]

\(6\)

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
    - 4-bit: 0010 = +2
    - 8-bit: 00000010 = +2
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**
  - Add 0’s?
  - 4-bit: `1100` = -4
  - 8-bit: `00001100` = +12
  - Make MSB 1?
  - 4-bit: `1100` = -4
  - 8-bit: `11111100` = -116
  - Add 1’s?
  - 4-bit: `1100` = -4
  - 8-bit: `11111100` = -4
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

```
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*
- **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 \(>>\)\(3\) and arithmetic shift is \(>>\)\(2\)
Shifting Arithmetic?

- What are the following computing?
  - x\(\gg n\):
    - \(0b\ 0100 \gg 1 = 0b\ 0010\)
    - \(0b\ 0100 \gg 2 = 0b\ 0001\)
    - Divide by \(2^n\)
  - x\(\ll n\):
    - \(0b\ 0001 \ll 1 = 0b\ 0010\)
    - \(0b\ 0001 \ll 2 = 0b\ 0100\)
    - 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:  $x \times 2^n$?

$x = 25; \quad 00011001 = \quad 25 \quad 25$

$L1=x<<2; \quad 0001100100 = \quad 100 \quad 100$

$L2=x<<3; \quad 00011001000 = \quad -56 \quad 200$

$L3=x<<4; \quad 000110010000 = \quad -112 \quad 144$

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:** $x / 2^n$

  - **Unsigned**
    - $x_u = 240u; \quad 11110000 = 240$
    - $R1_u = x_u >> 3; \quad 00011110000 = 30$
      - $240 / 2^3 = 30$
    - $R2_u = x_u >> 5; \quad 0000011110000 = 7$
      - $240 / 2^5 = 7.5$
      - 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:** \( x / 2^n \)?

\[\text{xs} = -16; \quad 11110000 = -16\]

\[\text{R1s=xu} >> 3; \quad 111111110000 = -2\]

\(-16/2^3 = -2\)

\[\text{R2s=xu} >> 5; \quad 11111111110000 = -1\]

\(-16/2^5 = -0.5\)

**rounding (down)**
Peer Instruction Question

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

- Assume we are using 8-bit arithmetic:
  - \( x \equiv (\text{unsigned char}) x \)
  - \( x \geq 128U \)
  - \( x \neq (x >> 2) \ll 2 \)
  - \( x == -x \)
    - Hint: there are two solutions
  - \( (x < 128U) \&\& (x > 0x3F) \)
Using Shifts and Masks

- Extract the 2\textsuperscript{nd} most significant \textit{byte} of an \textit{int}:
  - First shift, then mask: \((x\gg 16) \& 0xFF\)

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
x & 00000001 & 00000010 & 00000011 & 00000100 \\
\hline
x\gg 16 & 00000000 & 00000000 & 00000001 & 00000010 \\
\hline
0xFF & 00000000 & 00000000 & 00000000 & 11111111 \\
\hline
(x\gg 16) \& 0xFF & 00000000 & 00000000 & 00000000 & 00000010 \\
\hline
\end{array}
\]

Could also mask then shift in this case \(\rightarrow\) what could new mask be?
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 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 00000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(x)</th>
<th>10000010 00000100 00000111 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 00000000</td>
</tr>
<tr>
<td>((x\gg 31) &amp; 0x1)</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- Conditionals as Boolean expressions
  - For int x, what does \((x\ll\ll31)\gg31\) 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);`
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just interpret 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
  - Can be used in multiplication with constant or bit masking
  - Right shifting can be arithmetic (sign) or logical (0)