Integers II
CSE 351 Spring 2020

Instructor: 
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Connie Wang
Eddy (Tianyi) Zhou
Jonathan Chen
Millicent Li

Callum Walker
Diya Joy
Eric Fan
Joseph Schafer
Porter Jones

Chin Yeoh
Edan Sneh
Jeffery Tian
Melissa Birchfield
Rehaan Bhimani

http://xkcd.com/1953/
Administrivia

- hw4 due Friday – 11am
- hw5 due Monday – 11am
- Lab 1a due Monday (4/13)
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b coming soon, due 4/20
  - Bit puzzles on number representation
  - Can start after today’s lecture, but floating point will be introduced next week
  - Section worksheet for tomorrow has helpful examples
  - 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
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>= 0111</td>
<td>+7 ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>= 1111</td>
<td>-1 ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>+1101</td>
<td>-3</td>
</tr>
<tr>
<td>= 1001</td>
<td>+1 X</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:
  
  $\begin{cases}
  \text{bit representation of } x \\
  + \text{bit representation of } -x \\
  \hfill 0 \text{ (ignoring the carry-out bit)}
  \end{cases}$

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

  $\begin{array}{ccc}
  00000001 & + & \text{?? ?? ?? ?? ??} \\
  \hline
  00000000
  \end{array}$

  $\begin{array}{ccc}
  00000010 & + & \text{?? ?? ?? ?? ??} \\
  \hline
  00000000
  \end{array}$

  $\begin{array}{ccc}
  11000011 & + & \text{?? ?? ?? ?? ??} \\
  \hline
  00000000
  \end{array}$
Why Does Two’s Complement Work?

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

  (ignoring the carry-out bit)

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

  \[
  \begin{array}{ccc}
  00000001 & + & 11111111 \\
  \hline
  100000000 & + & 100000000
  \end{array}
  \]

  \[
  \begin{array}{ccc}
  00000010 & + & 11111110 \\
  \hline
  100000000 & + & 00111101
  \end{array}
  \]

  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

\[
2^{w-1} - 1 = 0b01\ldots1
\]

\[
-2^{w-1} = 0b10\ldots0 = \text{TMin}
\]

\[
\text{UMax} = 0b1\ldots1 = 2^w - 1
\]

\[
\text{UMax} - 1
\]

\[
\text{TMax} + 1
\]

\[
\text{TMax}
\]

\[
\text{TMin}
\]

\[
0b10\ldots0 = 2^{w-1}
\]

\[
= 0b0\ldots0 = \text{UMin}
\]
Values To Remember

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

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

- **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</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</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
- **Shifting and arithmetic operations** – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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 \(n\geq w\) \((w\) is 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?
  - $x >> n$
    - $0b\ 0100 \gg=\ 1 = 0b\ 0010$
    - $0b\ 0100 \gg=\ 2 = 0b\ 0001$
    - **Divide by $2^n$**
  - $x << n$
    - $0b\ 0001 \ll=\ 1 = 0b\ 0010$
    - $0b\ 0001 \ll=\ 2 = 0b\ 0100$
    - **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: \( x \times 2^n \)?

<table>
<thead>
<tr>
<th>( x = 25; )</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011001</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>L1 = ( x \ll 2 );</td>
<td>001100100</td>
<td>100 100</td>
</tr>
<tr>
<td>L2 = ( x \ll 3 );</td>
<td>0011001000</td>
<td>-56 200</td>
</tr>
<tr>
<td>L3 = ( x \ll 4 );</td>
<td>00110010000</td>
<td>-112 144</td>
</tr>
</tbody>
</table>

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: $x/2^n$?

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

$R1_u = x_u >> 3; \quad 00011110000 = 30$

$R2_u = x_u >> 5; \quad 0000011110000 = 7$

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:** $x/2^n$?

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

- Rounding (down)

\[ \frac{-16}{4} = -4 \]
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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</td>
<td>&gt;=</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>(unsigned) -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
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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: Unsigned

- **Addition:** drop carry bit \((-2^N)\)
  
  \[
  \begin{array}{c}
  15 \\
  + \ 2 \\
  \hline
  17 \\
  \end{array}
  \quad \begin{array}{c}
  1111 \\
  + \ 0010 \\
  \hline
  0001 \\
  \end{array}
  \]
  
  
- **Subtraction:** borrow \((+2^N)\)
  
  \[
  \begin{array}{c}
  1 \\
  - \ 2 \\
  \hline
  -1 \\
  \end{array}
  \quad \begin{array}{c}
  10001 \\
  - \ 0010 \\
  \hline
  1111 \\
  \end{array}
  \]

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

- **Addition:** $(+) + (+) = (-)$ result?
  
  $\begin{array}{c}
  6 \\
  + 3 \\
  \hline \\
  \textcolor{red}{9} \\
  \textcolor{red}{-7}
  \end{array}$

- **Subtraction:** $(-) + (-) = (+)$?
  
  $\begin{array}{c}
  -7 \\
  -3 \\
  \hline \\
  \textcolor{red}{-10} \\
  \textcolor{red}{6}
  \end{array}$

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
    - 4-bit: 0010 = +2
    - 8-bit: 00000010 = +2
  - Negative Case?
Polling Question [Int II - a]

- 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/reach](http://pollev.com/reach)

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…

\[
\begin{align*}
-8 & = -2^3 - 2^2 - 2^1 - 2^0 \\
-4 & = -2^2 - 2^1 - 2^0 \\
0 & = -2^0
\end{align*}
\]
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.
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$.

![Diagram showing sign extension process](attachment:sign_extension_diagram.png)
# Sign Extension Example

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

```java
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>
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
## Practice Question

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

### Assume we are using 8-bit arithmetic:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Example</th>
<th>All solutions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x == ) (unsigned char) ( x )</td>
<td>( x = 0 )</td>
<td>works for all ( x )</td>
</tr>
<tr>
<td>( x &gt;= 128 )</td>
<td>( x = -1 )</td>
<td>any ( x &lt; 0 )</td>
</tr>
<tr>
<td>( x != (x&gt;&gt;2) &lt;&lt; 2 )</td>
<td>( x = 3 )</td>
<td>any ( x ) where lowest two bits are not ( 00 )</td>
</tr>
<tr>
<td>( x == -x )</td>
<td>( x = 0 )</td>
<td>( 1 ) ( x = \text{ob}0\ldots0 = 0 ) ( 2 ) ( x = \text{ob}10\ldots0 = -128 )</td>
</tr>
<tr>
<td>( (x &lt; 128 ) &amp;&amp; ( (x &gt; 0\text{x3F}) )</td>
<td></td>
<td>any ( x ) where upper two bits are exactly ( \text{ob}01 )</td>
</tr>
</tbody>
</table>
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: \((x\gg16) \& \ 0xFF\)
  
  \[
  \begin{array}{c|cccccccc}
    x & 00000001 & 00000010 & 00000011 & 00000100 \\
    x\gg16 & 00000000 & 00000000 & 00000001 & 00000100 \\
    0xFF & 00000000 & 00000000 & 00000000 & 11111111 \\
    (x\gg16) \& 0xFF & 00000000 & 00000000 & 00000000 & 00000100 \\
  \end{array}
  \]

  - Or first mask, then shift: \((x \ & \ 0xFF0000) \gg16\)
  
  \[
  \begin{array}{c|cccccccc}
    x & 00000001 & 00000010 & 00000011 & 00000100 \\
    0xFF0000 & 00000000 & 11111111 & 00000000 & 00000000 \\
    x \ & \ 0xFF0000 & 00000000 & 00000010 & 00000000 & 00000000 \\
    (x\&0xFF0000) \gg16 & 00000000 & 00000000 & 00000000 & 00000100 \\
  \end{array}
  \]
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);`