Integers II
CSE 351 Spring 2019

Instructor: Ruth Anderson
Teaching Assistants:
- Gavin Cai
- Britt Henderson
- Sophie Tian
- Casey Xing
- Jack Eggleston
- Richard Jiang
- Connie Wang
- Chin Yeoh
- John Feltrup
- Jack Skalitzky
- Sam Wolfson

http://xkcd.com/1953/
Administrivia

- Homework 1 due TONIGHT (4/10)
  - Reminder: autograded, 20 tries, no late submissions
- Lab 1a due Monday (4/15)
  - Submit `pointer.c` and `lab1Areflect.txt` to Canvas
- Lab 1b released soon, due Monday 4/22
  - Bit puzzles on number representation
  - Have much of what you need after today, will need floating point, coming soon
  - Section tomorrow will be useful!
  - 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
  - Casting in C

- **Consequences of finite width representations**
  - Overflow, sign extension

- **Shifting and arithmetic operations**
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 $mod 2^w$

- **4-bit Examples:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
<td>4</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
<td>-3</td>
</tr>
<tr>
<td>=7</td>
<td>= -1</td>
<td></td>
<td>= 1</td>
<td></td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

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

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

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

\[
\begin{align*}
00000001 & + \ ???? ????? & = 00000000 \\
00000010 & + \ ???? ????? & = 00000000 \\
11000011 & + \ ???? ????? & = 00000000
\end{align*}
\]
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:

\[
\text{bit representation of } x + \text{bit representation of } -x + 0 \quad (\text{ignoring the carry-out bit})
\]

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

\[
\begin{align*}
00000001 & \quad + \quad 11111111 & \quad = \quad 1100000000 \\
00000010 & \quad + \quad 11111110 & \quad = \quad 1000000000 \\
11000011 & \quad + \quad 00111101 & \quad = \quad 1000000000 \\
\end{align*}
\]

These are the bitwise complement plus 1!

\[ -x == \sim x + 1 \]
Signed/Unsigned Conversion Visualized

- Two’s Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive
**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</td>
</tr>
<tr>
<td>Tmax</td>
<td>9,223,372,036,854,775,807</td>
<td>7F 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</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
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;`
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 mix 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>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0U</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0U</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647U</td>
<td></td>
<td>-2147483648</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648U</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648U</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</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

- 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</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</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</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</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
  \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
  \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
  9
  \end{array}
  \quad
  \begin{array}{c}
  0110 \\
  + 0011 \\
  \hline
  1001
  \end{array}
  \quad
  \begin{array}{c}
  -7
  \end{array}
  
- **Subtraction:** \((-)+(-)=(+)?\)
  
  \[
  \begin{array}{c}
  -7 \\
  - 3 \\
  \hline
  -10
  \end{array}
  \quad
  \begin{array}{c}
  1001 \\
  - 0011 \\
  \hline
  0110 \\
  \end{array}
  \quad
  \begin{array}{c}
  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 $\rightarrow$ short $\rightarrow$ int $\rightarrow$ long

- **4-bit $\rightarrow$ 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 \texttt{0b1100}?

- Underlined digit = MSB
- Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. \texttt{0b 0000 1100}
B. \texttt{0b 1000 1100}
C. \texttt{0b 1111 1100}
D. \texttt{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
- $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$

![Diagram showing sign extension](https://via.placeholder.com/150)
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 \(0\)s 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 \(0\)s 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
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></th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 25; )</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>( L1 = x \ll 2; )</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>( L2 = x \ll 3; )</td>
<td>-56</td>
<td>200</td>
</tr>
<tr>
<td>( L3 = x \ll 4; )</td>
<td>-112</td>
<td>144</td>
</tr>
</tbody>
</table>

- **Signed overflow**
- **Unsigned overflow**
Right Shifting 8-bit Examples

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

\[
xu = 240u; \quad 11110000 = 240
\]

\[
R1u=xu>>3; \quad 00011110000 = 30
\]

\[
R2u=xu>>5; \quad 0000011110000 = 7
\]

*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:** \( x / 2^n \)?

\[
\begin{align*}
x_s &= -16; \quad 11110000 &= -16 \\
R1_s &= x_u >> 3; \quad 11111110000 &= -2 \\
R2_s &= x_u >> 5; \quad 1111111110000 &= -1
\end{align*}
\]

rounding (down)
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:
  - $x == (\text{unsigned char}) \ x$
  - $x >= 128U$
  - $x != (x>>2) << 2$
  - $x == -x$
    - 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
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 2nd most significant byte of an int:
  - First shift, then mask: \((x >> 16) \& 0xFF\)
  - Or first mask, then shift: \((x \& 0xFF0000) >> 16\)
Using Shifts and Masks

- Extract the \textit{sign bit} of a signed \texttt{int}:
  - First shift, then mask: \( (x \gg 31) \& 0x1 \)
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear 1s possibly shifted in

\begin{tabular}{|c|c|}
\hline
\texttt{x} & \texttt{00000001 00000010 00000011 00000100} \\
\hline
\texttt{x\gg31} & \texttt{00000000 00000000 00000000 00000000} \\
\hline
\texttt{0x1} & \texttt{00000000 00000000 00000000 00000000} \\
\hline
\texttt{(x\gg31) \& 0x1} & \texttt{00000000 00000000 00000000 00000000} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\texttt{x} & \texttt{10000001 00000010 00000011 00000100} \\
\hline
\texttt{x\gg31} & \texttt{11111111 11111111 11111111 11111111} \\
\hline
\texttt{0x1} & \texttt{00000000 00000000 00000000 00000000} \\
\hline
\texttt{(x\gg31) \& 0x1} & \texttt{00000000 00000000 00000000 00000000} \\
\hline
\end{tabular}
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);`