Computer Systems
CSE 410 Autumn 2013
3 - Integers
Roadmap

C:
```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000001111
```

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac

Encoding
Integer & Floating Point Numbers

- Representation of integers: unsigned and signed
- Unsigned and signed integers in C
- Arithmetic and shifting
- Sign extension

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- Reading: Bryant/O’Hallaron sec. 2.2-2.3
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  - Interesting aside: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

- You add/subtract them using the normal “carry/borrow” rules, just in binary

- An important use of unsigned integers in C is pointers
  - There are no negative memory addresses
Signed Integers

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
  - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127

- But, we need to let about half of them be negative
  - Use the high order bit to indicate *negative*: call it the “sign bit”
    - Call this a “sign-and-magnitude” representation
  - Examples (8 bits):
    - 0x00 = 00000002 is non-negative, because the sign bit is 0
    - 0x7F = 011111112 is non-negative
    - 0x85 = 100001012 is negative
    - 0x80 = 100000002 is negative...
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - Sign-and-magnitude: 10000001_2
    Use the MSB for + or -, and the other bits to give magnitude
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- Sign-and-magnitude: 10000001₂

  Use the MSB for + or -, and the other bits to give magnitude
  (Unfortunate side effect: there are two representations of 0!)
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - Sign-and-magnitude: 10000001₂
    Use the MSB for + or -, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)
  - Another problem: math is cumbersome
    - Example:
      \[4 - 3 \neq 4 + (-3)\]

\[
\begin{array}{c|c}
0100 & +1011 \\
\hline
1111 & 1111
\end{array}
\]
Two’s Complement Negatives

- How should we represent -1 in binary?
  - Rather than a sign bit, let MSB have same value, but negative weight
    - W-bit word: Bits 0, 1, ..., W-2 add $2^0$, $2^1$, ..., $2^{W-2}$ to value of integer when set, but bit W-1 adds $-2^{W-1}$ when set
    - e.g. unsigned 1010₂: $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$
      2’s comp. 1010₂: $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$
  - So -1 represented as 1111₂; all negative integers still have MSB = 1
  - Advantages of two’s complement: only one zero, simple arithmetic
  - To get negative representation of any integer, take bitwise complement and then add one!
    \[ \sim x + 1 = -x \]
Two’s Complement Arithmetic

- The same addition procedure works for both unsigned and two’s complement integers
  - Simplifies hardware: only one adder needed
  - Algorithm: simple addition, discard the highest carry bit
    - Called “modular” addition: result is sum $modulo \ 2^w$

- Examples:

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>0100</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td>-3</td>
<td>+3</td>
</tr>
<tr>
<td>=7</td>
<td>0111</td>
<td>=1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drop carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 0001</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0011</th>
<th>0001</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=7</td>
<td></td>
<td>= 0001</td>
<td></td>
</tr>
</tbody>
</table>

Integers
Two’s Complement

- Why does it work?
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation.
  - This turns out to be the bitwise complement plus one.
    - What should the 8-bit representation of -1 be?
      - 00000001
      - +????????
      - 00000000 (we want whichever bit string gives the right result)

      00000010
      +????????
      00000000

      00000011
      +????????
      00000000
Two’s Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation.
  - This turns out to be the *bitwise complement plus one*.
    - What should the 8-bit representation of -1 be?
      
      \[
      \begin{array}{c}
      00000001 \\
      +11111111 \\
      \hline
      10000000
      \end{array}
      \]
      
      \[
      \begin{array}{c}
      00000010 \\
      +???????? \\
      \hline
      00000000
      \end{array}
      \]
      
      \[
      \begin{array}{c}
      00000011 \\
      +???????? \\
      \hline
      00000000
      \end{array}
      \]
Two’s Complement

- Why does it work?
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the bitwise complement plus one
    - What should the 8-bit representation of -1 be?
      - \[00000001\]
      - \[+11111111\] (we want whichever bit string gives the right result)
      - \[10000000\]

      \[00000010\] \[00000011\]
      - \[+11111110\] \[+11111101\]
      - \[10000000\] \[10000000\]
## Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>$X$</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>

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: $6 + 4 = ?$ $15U + 2U = ?$
  - If you compute a number that is too small, you wrap: $-7 - 3 = ?$ $0U - 2U = ?$
  - Answers are only correct mod $2^b$

- The CPU may be capable of “throwing an exception” for overflow on signed values
  - It won't for unsigned

- But C and Java just cruise along silently when overflow occurs...
Visualizations

- Same $W$ bits interpreted as signed vs. unsigned:

  Two’s complement

  $+2^{w-1}$

  $0$

  $-2^{w-1}$

  Unsigned

  $2^w$

  $2^{w-1}$

  $0$

  $-2^{w-1}$

- Two’s complement (signed) addition: $x$ and $y$ are $W$ bits wide

  $x + y$

  Positive overflow

  $+2^w$

  $+2^{w-1}$

  $0$

  $-2^{w-1}$

  Negative overflow

  $-2^w$

  $-2^{w-1}$

  Integers
Numeric Ranges

- **Unsigned Values**
  - \( U_{\text{Min}} = 0 \)
    - 000...0
  - \( U_{\text{Max}} = 2^w - 1 \)
    - 111...1

- **Two’s Complement Values**
  - \( T_{\text{Min}} = -2^{w-1} \)
    - 100...0
  - \( T_{\text{Max}} = 2^{w-1} - 1 \)
    - 011...1

- **Other Values**
  - Negative 1
    - 111...1 0xFFFFFFFF (32 bits)

**Values for \( W = 16 \)**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Integer & Floating Point Numbers

- Representation of integers: unsigned and signed
- Unsigned and signed integers in C
- Arithmetic and shifting
- Sign extension

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

- **Observations**
  - $|TMin| = TMax + 1$
  - Asymmetric range
  - $UMax = 2 * TMax + 1$

- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values are platform specific
  - See: `/usr/include/limits.h` on Linux
Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Use “U” suffix to force unsigned:
    - 0U, 4294967259U
Signed vs. Unsigned in C

- Casting
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - Explicit casting between signed & unsigned:
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments and function calls:
    - `tx = ux;`
    - `uy = ty;`
    - The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!
  - How does casting between signed and unsigned work – what values are going to be produced?
    - *Bits are unchanged*, just interpreted differently!
Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in a single expression, then **signed values implicitly cast to unsigned**
- Including comparison operations <, >, ==, <=, >=
- Examples for \( W = 32 \): \( TMIN = -2,147,483,648 \quad TMAX = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>Constant(_1)</th>
<th>Constant(_2)</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Integer & Floating Point Numbers

- Representation of integers: unsigned and signed
- Unsigned and signed integers in C
- Arithmetic and shifting
- Sign extension

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C
Shift Operations for unsigned integers

- **Left shift:** \( x << y \)
  - Shift bit-vector \( x \) left by \( y \) positions
  - Throw away extra bits on left
  - Fill with 0s on right

- **Right shift:** \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) positions
  - Throw away extra bits on right
  - Fill with 0s on left

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>&lt;&lt; 3</th>
<th>&gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>00000110</td>
<td>00110000</td>
<td>00000001</td>
</tr>
<tr>
<td>x</td>
<td>11110010</td>
<td>10010000</td>
<td>00111100</td>
</tr>
</tbody>
</table>

144 should be 1936
60 should be 60.5
### Shift Operations for signed integers

- **Left shift:** \( x << y \)
  - Equivalent to multiplying by \( 2^y \)
  - (if resulting value fits, no 1s are lost)

- **Right shift:** \( x >> y \)
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of \( x \)
  - Equivalent to dividing by \( 2^y \)
    - Correct rounding (towards 0) requires some care with signed numbers

### Examples

<table>
<thead>
<tr>
<th>( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Logical &gt;&gt; 2</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>Arithmetic &gt;&gt; 2</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Logical &gt;&gt; 2</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arithmetic &gt;&gt; 2</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>

*Undefined behavior when \( y < 0 \) or \( y \geq \text{word\_size} \)*
Using Shifts and Masks

- **Extract 2nd most significant byte of an integer**
  - First shift: \( x >> (2 \times 8) \)
  - Then mask: \( (x >> 16) \) & 0xFF

<table>
<thead>
<tr>
<th>x</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt;&gt; 16 )</td>
<td>00000000 00000000 01100001 01100010</td>
</tr>
<tr>
<td>( (x &gt;&gt; 16) ) &amp; 0xFF</td>
<td>\textcolor{red}{00000000 00000000 00000000 11111111} \textcolor{black}{00000000 00000000 01100010}</td>
</tr>
</tbody>
</table>

- **Extracting the sign bit**
  - \( (x >> 31) \) & 1 - need the “& 1” to clear out all other bits except LSB

- **Conditionals as Boolean expressions** (assuming \( x \) is 0 or 1)
  - if (x) \( a=y \) else \( a=z \); which is the same as \( a = x ? y : z \);
  - Can be re-written as: \( a = (x << 31) >> 31 \) & \( y + (!x << 31) >> 31 \) & \( z \);
Sign Extension

- **Task:**
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value

- **Rule:**
  - Make $k$ copies of sign bit:
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

\[ k \text{ copies of MSB} \]

\[ w \]
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension

```c
short int x = 12345;
int    ix = (int) x;
short int y = -12345;
int    iy = (int) y;
```

<table>
<thead>
<tr>
<th></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 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 01101101</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>