Agenda

• Review integer representations
  – How they show up in C
• Shifting / Masks
• Sign Extension
• Floating Point – more detail
How can we represent negative numbers?

• **Sign-and-Magnitude**
  – MSB denotes sign of number, rest of bits denote magnitude
  – E.g. 1001 = -1
  – Two zeros (0000 and 1000) and you need different hardware for + and –

• **One’s Complement (a.k.a. Bitwise Complement)**
  – Flip all bits to get the negative of a number
  – E.g. 1110 = -1
  – Two zeros (0000 and 11111)

• **Two’s Complement**
  – To get the negative of a number, flip all bits and add 1
  – E.g. 1111 = -1
  – Only one zero, can use same hardware for + and -, as well as for signed/unsigned
## Two’s complement

<table>
<thead>
<tr>
<th>Add</th>
<th>Invert and add</th>
<th>Invert and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>4 0100</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
<td>- 3 + 1101</td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td>= 1 1 0001</td>
</tr>
<tr>
<td></td>
<td>drop carry</td>
<td>= 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1 1111</td>
</tr>
</tbody>
</table>
Two’s Complement

• Why does it work?
  – The one’s complement of a b-bit positive number \( y \) is \((2^b - 1) - y\)
  • E.g. -3 = 1100₂ in one’s complement, which is 12 if it were unsigned
    \((2^4 - 1) - 3 = 12\)
  – Two’s Complement adds 1 to the one’s complement, thus -\( y \) is \( 2^b - y \) (or \(-x == (\sim x + 1)\))
    • \(-y\) and \( 2^b - y \) are equal mod \( 2^b \)
      (have the same remainder when divided by \( 2^b \))
    • Ignoring carries is equivalent to doing arithmetic mod \( 2^b \)
### Mapping Signed $\rightarrow$ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</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>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

$+16$
Signed vs. Unsigned in C

• Constants
  – Default = signed integers
  – Unsigned if they have “U” as a suffix
    • E.g. 0U, 1234567U
  – Size can by typed too
    • E.g. 1234567890123456ULL

• Casting

```c
int tx, ty;
unsigned ux, uy;

– Explicit casting
  tx = (int) ux;
  uy = (unsigned) ty;

– Implicit casting (careful!)
  tx = ux;
  uy = ty;
```
Casting Surprises

• If you mix unsigned and signed in a single expression, signed values are implicitly cast to unsigned
  – Including comparison operations <, >, ==, <=, >=

Examples for 32-bit: \( TMIN = -2,147,483,648 \) \( TMAX = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>Constant 1</th>
<th>Constant 2</th>
<th>Evaluated As</th>
<th>Relation between C1 and C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>Unsigned</td>
<td>==</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>Signed</td>
<td>&lt;</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>Unsigned</td>
<td>&gt;</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>Signed</td>
<td>&gt;</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td>Unsigned</td>
<td>&lt;</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>Signed</td>
<td>&gt;</td>
</tr>
<tr>
<td>0U – 1</td>
<td>-2</td>
<td>Unsigned</td>
<td>&gt;</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>Unsigned</td>
<td>&lt;</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>Signed</td>
<td>&gt;</td>
</tr>
</tbody>
</table>
Shift Operations

- **Left Shift**: \( x << y \)
  - Shift bit-vector \( x \) left by \( y \) positions
  - Throw away extra bits on left, fill with 0’s on right
  - Each shift left by 1 bit is the same as multiplying by 2
    - So \( x << y \) is the same as \( x \times 2^y \)

- **Right Shift**: \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) positions
  - Throw away extra bits on right
    - Logical shift (for unsigned): Fill with 0’s on left
    - Arithmetic shift (for signed): Fill with whatever was MSB on left – Maintain the sign of \( x \)
  - Each shift right by 1 is the same as dividing by 2
Masking

• What if you need to extract the 2\textsuperscript{nd} most significant byte of an integer (i.e. bits 16 through 23)?
  – First shift: $x >> 16$
  – Then mask: $(x >> 16) \& 0xff$

\begin{tabular}{|c|c|}
  \hline
  x & 01100001 01100010 01100011 01100100 \\
  \hline
  x >> 16 & 00000000 00000000 01100001 01100010 \\
  \hline
  (x >> 16) & 0xff & 00000000 00000000 00000000 11111111 \\
  \hline
  ___ & 00000000 00000000 00000000 01100010 \\
  \hline
\end{tabular}

• Extracting the sign bit
  – $(x >> 31) \& 1$
  – Need the “& 1” to clear out all other bits except the LSB
Sign Extension

- Given a $w$-bit signed integer $x$, convert to a $(w+k)$-bit signed integer with the same value.
- Rule: Make $k$ copies of sign bit.

\[-X_2 = X_{w-1},\ldots,X_{w-1},X_{w-1},X_{w-2},\ldots,X_0\]

\[X'_2 = X_k,\ldots,X_k,\ldots,\ldots,\ldots,\ldots,X_k\]
Sign Extension Example

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>

C automatically performs sign extension
Fractional Binary Numbers (Not Floating Point!)

Bits to right of “binary point” represent fractional powers of 2
Fractional Binary Numbers Examples

• What are these numbers in binary?
  • 5 and \(\frac{3}{4}\)  \(101.11_2\)
  • 2 and 7/8  \(10.111_2\)
  • 63/64  \(0.111111_2\)

• Observations
  • Divide by 2 by shifting right
  • Multiply by 2 by shifting left
  • Numbers of form \(0.111111\ldots_2\) are just below 1.0
Representable Numbers

• Limitation
  – Can only exactly represent numbers of the form $\frac{x}{2^k}$
  – Other rational numbers have repeating bit representations

• Value  Representation
  1/3  0.010101010101[01]...$_2$
  1/5  0.001100110011[0011]...$_2$
  1/10 0.0001100110011[0011]...$_2$
Fixed Point Representation

• Pick where you want to put the decimal point
• The position of the binary point affects the **range** and **precision**
  – Range: difference between the largest and smallest representable numbers
  – Precision: smallest possible difference between any two numbers
• **Pro**
  – Simple: The same hardware that does integer arithmetic can do fixed point arithmetic
    • In fact, the programmer can use ints with an implicit fixed point
      – E.g. int balance; // number of pennies in the account
    • ints are just fixed point numbers with the binary point to the right of the LSB
• **Con**
  – There is no good way to pick where the fixed point should be
    • Sometimes you need range, sometimes you need precision
    • More range = less precision and vice versa
Floating Point Representation

\[ (-1)^S \times M \times 2^E \]

- Sign bit \( S \) determines whether number is negative or positive
- Mantissa \( M \) (aka Significand aka “Frac”) normally a fractional value in range \([1.0, 2.0)\).
- Exponent \( E \) weights value by power of two
- Encoding
  - MSB is sign bit \( S \)
  - frac field encodes \( M \) (but is not equal to \( M \))
  - exp field encodes \( E \) (but is not equal to \( E \))
Precisions

- Single precision (float): 32 bits
  \[ s \quad \text{exp} \quad \frac{\text{frac}}{} \]
  
  \[
  1 \quad 8 \quad 23
  \]

- Double Precision (double): 64 bits
  \[ s \quad \text{exp} \quad \frac{\text{frac}}{} \]
  
  \[
  1 \quad 11 \quad 52
  \]

- Extended Precision: 80 bits (Intel only)
  \[ s \quad \text{exp} \quad \frac{\text{frac}}{} \]
  
  \[
  1 \quad 15 \quad 64
  \]
Normalization, Bias and Special Values

• “Normalized” means mantissa has form 1.xxxx
  – 0.011 * 2^5 and 1.1 * 2^3 represent the same number, but the latter makes better use of available bits
  – Since we know the mantissa starts with a 1, don’t bother to store it
  – Therefore, when the mantissa is 1.xxxxx, M (i.e. \( \frac{}{} \)) contains xxxxx
• The exponent field does not contain the exponent of the number, but the offset from a bias
  – \( \exp = E + \text{Bias} \)
  – \( \text{Bias} = 2^{|\exp|-1} - 1 \) where \( |\exp| = \text{size of } \exp \text{ field} \)
  – (e.g. 127 is the bias for an 8 bit \( \exp \))
• Special Values
  – The float value 00....0 represents zero
  – Exp = 11...1 and Mantissa = 00...0 represents infinity
    • E.g. 10.0 / 0.0
  – Exp = 11...1 and Mantissa != 00...0 represents NaN
    • E.g. 0 * Infinity
Floating Point Example

• How is float 12345.0 represented?

• Value

\[ 12345.0_{10} = 11000000111001_2 \]

\[ = 1.1000000111001_2 \times 2^{13} \]
Floating Point Example

<table>
<thead>
<tr>
<th>s</th>
<th>exp (8)</th>
<th>frac (23)</th>
</tr>
</thead>
</table>

- How is float 12345.0 represented?
- Value
  \[
  12345.0_{10} = 11000000111001_2
  = 1.1000000111001_2 \times 2^{13}
  \]
- Mantissa
  \[
  M = 1.1000000111001_2
  frac = 1000000111001000000000000_2 \text{ (Need to extend to fill all 23 bits)}
  \]
Floating Point Example

• How is float 12345.0 represented?

• Value

\[
12345.0_{10} = 11000000111001_2
\]

\[
= 1.1000000111001_2 * 2^{13}
\]

• Mantissa

\[
M = 1.1000000111001_2
\]

\[
frac = 1000000110010000000000002 \quad \text{(Need to extend to fill all 23 bits)}
\]

• Exponent

\[
E = 13
\]

\[
Bias = 2^7 - 1 = 127
\]

\[
exp = 140_{10} = 10001100_2
\]
Floating Point Operations

• Basic Idea
  – First compute exact result
  – Make it fit into desired precision
    • Possibly overflow if exponent is too large
    • Possibly round to fit into $\text{frac}$

• $x +_f y = \text{Round}(x + y)$
• $x *_f y = \text{Round}(x * y)$
Floating Point Multiplication

\[ (-1)^{S_1} M_1 \ 2^{E_1} \ * \ (-1)^{S_2} M_2 \ 2^{E_2} \]

• Exact Result
  – Sign = \( S_1 \ \text{^} \ S_2 \)
  – Mantissa: \( M_1 \ * \ M_2 \)
  – Exponent: \( E_1 + E_2 \)

• Fixing
  – If \( M \geq 2 \), \( M = M \gg 1 \), \( E = E + 1 \)
  – If \( E \) is out of range, overflow
  – Round \( M \) to fit \( \text{frac} \) precision
Floating Point Addition

\[ (-1)^{S_1} M_1 2^{E_1} + (-1)^{S_2} M_2 2^{E_2} \]

Assume \( E_1 > E_2 \)

**Exact Result**
- Sign \( S \), Mantissa \( M \):
  - Shift sign and mantissa of first value left by the difference of the exponents. This makes exponents equal, so you can add the signed mantissas.
- Exponent \( E \): \( E_1 \)

**Fixing**
- If \( M \geq 2 \), \( M = M >> 1 \), \( E = E + 1 \)
- If \( M < 1 \), \( M = M << k \), \( E = E - k \)
- Overflow if \( E \) is out of range
- Round \( M \) to fit \( \frac{\text{frac precision}}{\text{precision}} \)
Rounding Errors

• Since we round on every operation, the operations are not really associate or distributive

  — Let a = 1.52342, b = 6.2342342, c = 2.2523555

  • (a + b) + c = 10.010009700000001
    a + (b + c) = 10.010009699999999

  • a * (b + c) = 12.928640480774000
    a * b + a * c = 12.928640480774002
Floating Point Values and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for ( i=0; i<10; i++ ) {
        f2 += 1.0/10.0;
    }
    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 == f2? %s\n", f1 == f2 ? "yes" : "no");
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n\n", f2);
    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    return 0;
}
```
Floating Point Values and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i=0; i<10; i++) {
        f2 += 1.0/10.0;
    }
    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 == f2? %s\n", f1 == f2 ? "yes" : "no");
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n", f2);
    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    return 0;
}
```

$ ./a.out
0x3f800000  0x3f800001
f1 == f2? no
f1 = 1.00000000
f2 = 1.000000119
f1 == f3? yes
Summary

• As with integers, floats suffer from the fixed number of bits available to represent them
  – Can get overflow/underflow, just like ints
  – Some “simple fractions” have no exact representation (e.g. 0.1)
  – Can also lose precision, unlike ints
    • “Every operation gets a slightly wrong result”

• Mathematically equivalent ways of writing an expression may compute different results

• NEVER test floating point values for equality!