Computer Systems
CSE 410 Autumn 2013
4 – Floating Point
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. 2.4
Fractional Binary Numbers

- What is $1011.101_2$?

- How do we interpret fractional *decimal* numbers?
  - e.g. $107.95_{10}$
  - Can we interpret fractional binary numbers in an analogous way?
**Fractional Binary Numbers**

- **Representation**
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \( \sum_{k=-j}^{i} b_k \cdot 2^k \)
Fractional Binary Numbers: Examples

- **Value**
  - 5 and 3/4
  - 2 and 7/8
  - 63/64

- **Representation**
  - 101.11₂
  - 10.111₂
  - 0.111111₂

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of the form 0.111111...₂ are just below 1.0
    - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... → 1.0
    - Shorthand notation for all 1 bits to the right of binary point: 1.0 – ε
Representable Values

- Limitations of fractional binary numbers:
  - Can only exactly represent numbers that can be written as $x \times 2^y$
  - Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.010101010101010101</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011001100</td>
</tr>
<tr>
<td>1/10</td>
<td>0.000110011001100110</td>
</tr>
</tbody>
</table>
Fixed Point Representation

- We might try representing fractional binary numbers by picking a fixed place for an implied binary point
  - “fixed point binary numbers”
- Let's do that, using 8-bit fixed point numbers as an example
  - #1: the binary point is between bits 2 and 3
    \[ b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ [. \ b_2 \ b_1 \ b_0 \]
  - #2: the binary point is between bits 4 and 5
    \[ b_7 \ b_6 \ b_5 \ [. \ b_4 \ b_3 \ b_2 \ b_1 \ b_0 \]
- The position of the binary point affects the range and precision of the representation
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
Fixed Point Pros and Cons

■ Pros
  ▪ It's 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
    ▪ ints are just fixed point numbers with the binary point to the right of $b_0$

■ Cons
  ▪ There is no good way to pick where the fixed point should be
    ▪ Sometimes you need range, sometimes you need precision – the more you have of one, the less of the other.
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
IEEE Floating Point

- **Analogous to scientific notation**
  - Not 12000000 but 1.2 x 10^7; not 0.0000012 but 1.2 x 10^{-6}
  - (write in C code as: 1.2e7; 1.2e-6)

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs today

- **Driven by numerical concerns**
  - Standards for handling rounding, overflow, underflow
  - Hard to make fast in hardware but numerically well-behaved

- **1989 Turing Award to William Kahan (UC Berkeley)**
Floating Point Representation

**Numerical form:**

\[ V_{10} = (-1)^s \times M \times 2^E \]

- Sign bit \( s \) determines whether number is negative or positive
- Significand (mantissa) \( M \) normally a fractional value in range \([1.0,2.0)\)
- Exponent \( E \) weights value by a (possibly negative) power of two

**Representation in memory:**

- MSB \( s \) is sign bit \( s \)
- \( \text{exp} \) field encodes \( E \) (but is *not equal* to \( E \))
- \( \text{frac} \) field encodes \( M \) (but is *not equal* to \( M \))
Precisions

- **Single precision: 32 bits**
  
  
  ![Single precision diagram]
  
  1  k=8  n=23

- **Double precision: 64 bits**
  
  ![Double precision diagram]
  
  1  k=11  n=52
Normalization and Special Values

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

- “Normalized” means the mantissa \( M \) has the form 1.xxxxx
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it!

- How do we represent 0.0? Or special / undefined values like 1.0/0.0?

IEEE Floating Point Standard
Normalization and Special Values

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

- **“Normalized”** means the mantissa \( M \) has the form \( 1.xxxxx \)
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **Special values:**
  - The bit pattern 00...0 represents zero
  - If \( \text{exp} == 11...1 \) and \( \text{frac} == 00...0 \), it represents \( \infty \)
    - e.g. \( 1.0/0.0 = -1.0/-0.0 = +\infty \), \( 1.0/-0.0 = -1.0/0.0 = -\infty \)
  - If \( \text{exp} == 11...1 \) and \( \text{frac} \neq 00...0 \), it represents NaN: “Not a Number”
    - Results from operations with undefined result,
      e.g. \( \sqrt{-1} \), \( \infty - \infty \), \( \infty \times 0 \)

IEEE Floating Point Standard
Normalized Values

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

- **Condition:** \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)
- **Exponent coded as biased value:** \( E = \text{exp} - \text{Bias} \)
  - \( \text{exp} \) is an unsigned value ranging from 1 to \( 2^{k-2} \) (\( k = \# \) bits in \( \text{exp} \))
  - \( \text{Bias} = 2^{k-1} - 1 \)
    - Single precision: 127 (so \( \text{exp} \): 1\ldots254, \( E \): -126\ldots127)
    - Double precision: 1023 (so \( \text{exp} \): 1\ldots2046, \( E \): -1022\ldots1023)
  - These enable negative values for \( E \), for representing very small values

- **Significand coded with implied leading 1:** \( M = 1.\text{xxx}\ldots\text{x} \)
  - \( \text{xxx}\ldots\text{x} \): the \( n \) bits of \( \text{frac} \)
  - Minimum when \( 000\ldots0 \) (\( M = 1.0 \))
  - Maximum when \( 111\ldots1 \) (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”
Normalized Encoding Example

\[ V = (-1)^s \times M \times 2^E \]

- **Value:** \( \text{float } f = 12345.0; \)
  - \( 12345_{10} = 11000000111001_2 \)
    - \( = 1.1000000111001_2 \times 2^{13} \) (normalized form)

- **Significand:**
  - \( M = 1.1000000111001_2 \)
  - \( \text{frac} = 10000001110010000000000000_2 \)

- **Exponent:** \( E = \text{exp} - \text{Bias}, \text{ so } \exp = E + \text{Bias} \)
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \exp = 140 = 10001100_2 \)

- **Result:**
  - \( \begin{array}{ccc}
  0 & 10001100 & 10000001110010000000000000 \\
  s & \text{exp} & \text{frac}
  \end{array} \)
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
How do we do operations?

- Unlike the representation for integers, the representation for floating-point numbers is not *exact*. 
Floating Point Operations: Basic Idea

\[ V = (-1)^s \times M \times 2^E \]

- \( x \_f \ y = \text{Round}(x + y) \)
- \( x \_f \ y = \text{Round}(x \times y) \)

Basic idea for floating point operations:
- First, compute the exact result
- Then, round the result to make it fit into desired precision:
  - Possibly overflow if exponent too large
  - Possibly drop least-significant bits of significand to fit into \( \text{frac} \)
Rounding modes

- Possible rounding modes (illustrated with dollar rounding):

<table>
<thead>
<tr>
<th></th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>−$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-toward-zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>−$1</td>
</tr>
<tr>
<td>Round-down (−∞)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>−$2</td>
</tr>
<tr>
<td>Round-up (+∞)</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>−$1</td>
</tr>
<tr>
<td>Round-to-nearest</td>
<td>$1</td>
<td>$2</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>Round-to-even</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>−$2</td>
</tr>
</tbody>
</table>

- What could happen if we’re repeatedly rounding the results of our operations?
  - If we always round in the same direction, we could introduce a statistical bias into our set of values!

- Round-to-even avoids this bias by rounding up about half the time, and rounding down about half the time
  - Default rounding mode for IEEE floating-point
Mathematical Properties of FP Operations

- If overflow of the exponent occurs, result will be $\infty$ or $-\infty$
- Floats with value $\infty$, $-\infty$, and NaN can be used in operations
  - Result is usually still $\infty$, $-\infty$, or NaN; sometimes intuitive, sometimes not

- Floating point operations are not always associative or distributive, due to rounding!
  - $(3.14 + 1e10) - 1e10 \neq 3.14 + (1e10 - 1e10)$
  - $1e20 \times (1e20 - 1e20) \neq (1e20 \times 1e20) - (1e20 \times 1e20)$
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
Floating Point in C

- C offers two levels of precision
  - `float` single precision (32-bit)
  - `double` double precision (64-bit)
- Default rounding mode is round-to-even
- `#include <math.h>` to get INFINITY and NAN constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!
  - Substitute things like: `(abs(x-y) / max(abs(x),abs(y))) < epsilon`
    - Not guaranteed – depends on what you’re doing, the data, etc.
    - When in doubt, find a trained Numerical Analyst or use a carefully-written library
Floating Point in C

Conversions between data types:

- Casting between int, float, and double changes the bit representation!!
- int $\rightarrow$ float
  - May be rounded; overflow not possible
- int $\rightarrow$ double or float $\rightarrow$ double
  - Exact conversion, as long as int has $\leq$ 53-bit word size
- double or float $\rightarrow$ int
  - Truncates fractional part (rounded toward zero)
  - Not defined when out of range or NaN: generally sets to Tmin
Summary

- **Zero**
  
  | s | 0 | 00000000 | 00000000000000000000000000000000 |

- **Normalized values**

  | s | 1 to $2^k - 2$ | significand = 1.M |

- **Infinity**

  | s | 11111111 | 000000000000000000000000000000000 |

- **NaN**

  | s | 11111111 | non-zero |

- **Denormalized values**

  | s | 00000000 | significand = 0.M |
Summary (cont’d)

- 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., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute different results
  -Violates associativity/distributivity

- Never test floating point values for equality!