The Hardware/Software Interface
CSE351 Winter 2013

Floating-Point Numbers

Roadmap

C:
```c
char *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:
```assembly
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret
```

Machine code:
```
0111010000011000
1000110100000100
1000100111000010
1100000011110101

0111010000011000
1000110100000100
1000100111000010
1100000011110101
```

Computer system:
```
Windows 8
Mac
```

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

Fractional Binary Numbers
- What is 1011.101₂?
- How do we interpret fractional decimal numbers?
  - e.g. 107.95₁₀
  - Can we interpret fractional binary numbers in an analogous way?
Fractional Binary Numbers

- Bits to right of “binary point” represent fractional powers of 2.
- Represents rational number: \( \sum_{k=-j}^{j} b_k \cdot 2^k \)

Representable Values

- Limitations of fractional binary numbers:
  - Can only exactly represent numbers that can be written as \( x \cdot 2^y \)
  - 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>0.0101010101 [01] (_2)</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011 [0011] (_2)</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011 [0011] (_2)</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:

<table>
<thead>
<tr>
<th>#1: the binary point is between bits 2 and 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_2 b_1 b_0 [ ] b_1 b_2 b_3 )</td>
</tr>
<tr>
<td>#2: the binary point is between bits 4 and 5</td>
</tr>
<tr>
<td>( b_5 b_4 b_3 [ ] b_2 b_1 b_0 )</td>
</tr>
</tbody>
</table>

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

Observations:

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of the form \( 0.111111\ldots \_2 \) are just below 1.0
  - \( 1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0 \)
  - Shorthand notation for all 1 bits to the right of binary point: \( 1.0 - \varepsilon \)
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.

---

IEEE Floating Point

**Analogous to scientific notation**
- Not 12000000 but $1.2 \times 10^7$; not 0.0000012 but $1.2 \times 10^{-6}$
  - (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
  - Numerical analysts predominated over hardware designers in defining standard

---

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 $s$
- $exp$ field encodes $E$ (but is **not equal** to $E$)
- $frac$ field encodes $M$ (but is **not equal** to $M$)

---

Precisions

**Single precision:** 32 bits

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k=8</td>
<td>n=23</td>
</tr>
</tbody>
</table>

**Double precision:** 64 bits

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k=11</td>
<td>n=52</td>
</tr>
</tbody>
</table>
Normalization and Special Values

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

- **Normalized** means the mantissa \( M \) has the form \( 1.xxxxx \)
  - 0.011 \( x \) and 1.1 \( x \) 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?

**Normalized Values**

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

- **Condition:** \( \exp \neq 000...0 \) and \( \exp \neq 111...1 \)
- **Exponent coded as biased value:** \( E = \exp - \text{Bias} \)
  - \( \exp \) is an **unsigned** value ranging from 1 to \( 2^k - 2 \) (\( k = \# \) bits in \( \exp \))
  - **Bias** = \( 2^{k-1} - 1 \)
    - **Single precision:** 127 (so \( \exp \): 1...254, \( E \): -126...-127)
    - **Double precision:** 1023 (so \( \exp \): 1...2046, \( E \): -1022...1023)
  - These enable negative values for \( E \), for representing very small values

- **Significand coded with implied leading 1:** \( M = 1.xxxxxx \)
  - \( xxx...x \): the \( n \) bits of \( \frac{}{} \)
  - Minimum when 000...0 (\( M = 1.0 \))
  - Maximum when 111...1 (\( M = 2.0 - \epsilon \))
  - Get extra leading bit for “free”

**Normalized Encoding Example**

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

- **Value:** \( f = 12345 \cdot 0 \)\( \_ \)
  - \( 12345_{10} = 11000000111001_2 \)
  - \( 1.1000000111001_2 \cdot 2^{13} \) (normalized form)

- **Significand:**
  - \( M = 1.1000000111001 \)
  - \( \text{frac} = 10000001110010000000000_2 \)

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

- **Result:**
  - \( 010001100 \_ \)
  - \( 10000001110010000000000 \)
  - **s exp frac**
How do we do operations?

- Unlike the representation for integers, the representation for floating-point numbers is not exact.

Floating Point Operations: Basic Idea

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

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

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 frac

Rounding modes

- Possible rounding modes (illustrate with dollar rounding):
  - Round-toward-zero
  - Round-down (\(-\infty\))
  - Round-up (\(+\infty\))
  - Round-to-nearest
  - Round-to-even

- 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

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!
  - \((3.14 + 1e10) - 1e10 \neq 3.14 + (1e10 - 1e10)\)
  - \(1e20 \times (1e20 - 1e20) \neq (1e20 \times 1e20) - (1e20 \times 1e20)\)
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 INFFINITY and NaN constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  
  - Just avoid them!

Floating Point in C

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

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., 0.2)
  
  - 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!

Additional details

- Denormalized values – to get finer precision near zero

- Tiny floating point example

- Distribution of representable values

- Floating point multiplication & addition

- Rounding
Denormalized Values

- **Condition**: \( \exp = 000...0 \)

- Exponent value: \( E = \exp - \text{Bias} + 1 \) (instead of \( E = \exp - \text{Bias} \))

- Significand coded with implied leading 0: \( M = 0 . \text{xxx}...x \)
  - \text{xxx}...\text{x} bits of \( \text{frac} \)

- **Cases**
  - \( \exp = 000...0, \frac{\text{frac}}{} = 000...0 \)
    - Represents value 0
    - Note distinct values: +0 and −0 (why?)
  - \( \exp = 000...0, \frac{\text{frac}}{} \neq 000...0 \)
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Special Values

- **Condition**: \( \exp = 111\ldots1 \)

- **Case**: \( \exp = 111\ldots1, \frac{\text{frac}}{} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -1.0/0.0 = -\infty \)

- **Case**: \( \exp = 111\ldots1, \frac{\text{frac}}{} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty = \infty, \infty * 0 \)

Visualization: Floating Point Encodings

Tiny Floating Point Example

\[
\begin{array}{c|c|c}
s & \exp & \frac{\text{frac}}{} \\
\hline
& 1 & 4 \\
\end{array}
\]

- **8-bit Floating Point Representation**
  - The sign bit is in the most significant bit.
  - The next four bits are the exponent, with a bias of 7.
  - The last three bits are the \( \text{frac} \)

- **Same general form as IEEE Format**
  - Normalized, denormalized
  - Representation of 0, NaN, infinity
### Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>$s$</th>
<th>$\text{exp}$</th>
<th>$\text{frac}$</th>
<th>$\ell$</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>001</td>
<td>-6</td>
<td>$1/8^*1/64 = 1/512$</td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>010</td>
<td>-6</td>
<td>$2/8^*1/64 = 2/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td>-6</td>
<td>$6/8^*1/64 = 6/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>-6</td>
<td>$7/8^*1/64 = 7/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>-6</td>
<td>$2/8^*1/64 = 2/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
<td>-6</td>
<td>$8/8^*1/64 = 8/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001</td>
<td>-6</td>
<td>$9/8^*1/64 = 9/512$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010</td>
<td>-6</td>
<td>closest to zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td>-6</td>
<td>largest denorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>-1</td>
<td>$14/8^*1/2 = 14/16$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>-1</td>
<td>$15/8^*1/2 = 15/16$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
<td>-6</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001</td>
<td>-6</td>
<td>$9/8^*1 = 9/8$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010</td>
<td>-6</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td>-6</td>
<td>$10/8^*1 = 10/8$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>7</td>
<td>$14/8^*128 = 224$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>7</td>
<td>$15/8^*128 = 240$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

### Distribution of Values

- **6-bit IEEE-like format**
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is 3

- **Interesting Numbers**

  **Zero**
  - Single = $1.4 \times 10^{-38}$
  - Double = $4.9 \times 10^{-324}$

  **Smallest Pos. Denorm.**
  - Single = $1.18 \times 10^{-38}$
  - Double = $2.2 \times 10^{-308}$

  **Largest Denormalized**
  - Single = $1.18 \times 10^{38}$
  - Double = $2.2 \times 10^{308}$

  **Smallest Pos. Norm.**
  - Just larger than largest denormalized
  - Single = $3.4 \times 10^{-38}$
  - Double = $1.8 \times 10^{308}$

  **One**
  - 01111

  **Largest Normalized**
  - Double = $2 \times 10^{307}$

### Distribution of Values (close-up view)

#### 6-bit IEEE-like format
- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is 3

#### Interesting Numbers
- **Zero**
  - Single = $1.4 \times 10^{-38}$
  - Double = $4.9 \times 10^{-324}$

- **Smallest Pos. Denorm.**
  - Single = $1.18 \times 10^{-38}$
  - Double = $2.2 \times 10^{-308}$

- **Largest Denormalized**
  - Single = $1.18 \times 10^{38}$
  - Double = $2.2 \times 10^{308}$

- **Smallest Pos. Norm.**
  - Just larger than largest denormalized
  - Single = $3.4 \times 10^{-38}$
  - Double = $1.8 \times 10^{308}$

- **One**
  - 01111

- **Largest Normalized**
  - Double = $2 \times 10^{307}$
Special Properties of Encoding

- Floating point zero (0) exactly the same bits as integer zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider 0 = 0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Multiplication

\((-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}\)

- Exact Result: \((-1)^s M 2^E\)
  - Sign s:
    - \(s_1 \land s_2 \quad //\ xor \ of\ s1\ and\ s2\)
  - Significand M:
    - \(M_1 \ast M_2\)
  - Exponent E:
    - \(E_1 + E_2\)
- Fixing
  - If \(M \geq 2\), shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision

Floating Point Addition

\((-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}\)

- Exact Result: \((-1)^s M 2^E\)
  - Assume \(E_1 > E_2\)
  - \((-1)^{s_1} M_1\)
  - \((-1)^{s_2} M_2\)
  - \((-1)^s M\)
- Fixing
  - If \(M \geq 2\), shift M right, increment E
  - If \(M < 1\), shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit frac precision

Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under- estimated
- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 1.2349999 → 1.23 (Less than half way)
    - 1.2350001 → 1.24 (Greater than half way)
    - 1.2350000 → 1.24 (Half way—round up)
    - 1.2450000 → 1.24 (Half way—round down)
Rounding Binary Numbers

- Binary Fractional Numbers
  - "Half way" when bits to right of rounding position = 100...

Examples
- Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/32</td>
<td>10.00011, 10.001</td>
<td>&lt;1/2—down</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2/16</td>
<td>10.00110, 10.010</td>
<td>&gt;1/2—up</td>
<td>2 1/4</td>
<td></td>
</tr>
<tr>
<td>2/8</td>
<td>10.11000, 11.000</td>
<td>1/2—up</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2/5</td>
<td>10.10100, 10.100</td>
<td>1/2—down</td>
<td>2 1/2</td>
<td></td>
</tr>
</tbody>
</table>

Floating Point 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
", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f
", f1);
    printf("f2 = %10.8f

", f2);
    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s
", f1 == f3 ? "yes" : "no" );
    return 0;
}
```

`$ ./a.out`

0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes

Memory Referencing Bug

```c
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.139999864856
fun(3) -> 2.00000001035156
fun(4) -> 3.14, then segmentation fault

Explanation:
- Saved State
- Location accessed by fun(i)

Representing 3.14 as a Double FP Number

- 1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000
- 3.14 = 11.0010 0011 1101 0111 0010 0000...
- (-1) x \(2^{11}\)
  - S = 0 encoded as 0
  - M = 1.1001 0001 1110 1111 1000 1001 0000.... (leading 1 left out)
  - E = 1 encoded as 1024 (with bias)

s  exp (11) frac (first 20 bits)
0 100 0000 0000 1001 0001 1110 1111 1000

frac (the other 32 bits)
0101 0000...
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0)  ->  3.14
fun(1)  ->  3.14
fun(2)  ->  3.1399998664856
fun(3)  ->  2.00000061035156
fun(4)  ->  3.14, then segmentation fault

Saved State
    Location accessed by fun(i)
    d7 ... d4 0100 0000 0000 1001 0001 1110 1011 1000
    d3 ... d0 0101 0000 ...
    a[1] 1
    a[0] 0

Fun(0)  ->  3.14
Fun(1)  ->  3.14
Fun(2)  ->  3.1399998664856
Fun(3)  ->  2.00000061035156
Fun(4)  ->  3.14, then segmentation fault

Saved State
    Location accessed by fun(i)
    d7 ... d4 0100 0000 0000 1001 0001 1110 1011 1000
    d3 ... d0 0100 0000 0000 0000 0000 0000 0000 0000
    a[1] 1
    a[0] 0