Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective.” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Byte-Oriented Memory Organization

- Programs refer to data by address
  - conceptually, envision it as a very large array of bytes
    » in reality, it's not, but can think of it that way
  - an address is like an index into that array
    » pointer variables contain addresses

- Note: system provides private address spaces to each "process"
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others
Machine Words

• Any given computer has a “word size”
  – nominal size of integer-valued data
    » and of addresses

  – until a decade or so ago, most machines used 32 bits (4 bytes) as word size
    » limits addresses to 4GB ($2^{32}$ bytes)
    » became too small for memory-intensive applications
      • leading to emergence of computers with 64-bit word size

  – machines still support multiple data formats
    » fractions or multiples of word size
    » always integral number of bytes
**Word-Oriented Memory Organization**

- **Addresses specify byte locations**
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0000</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0000</td>
<td>0003</td>
<td>0003</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Read “Gulliver’s Travels” for an explanation of the egg.
Here we have a four-byte integer one. In the big-endian representation, the address of the integer is the address of the byte containing its most-significant bits (the big end), while in the little-endian representation, the address of the integer is the address of the byte containing its least-significant bits (the little end). Suppose we pass a pointer to this integer to some procedure. However, in a type-mismatch, the procedure assumes that what is passed it is a two-byte integer. On a big-endian system, it would think it was passed a zero, but on a little-endian system, it would think it was passed a one.

This is not an argument in favor of either approach, but simply an observation that behaviors could be different.
Quiz 1

```c
int main() {
  long x=1;
  func((int *)&x);
  return 0;
}

void func(int *arg) {
  printf("%d\n", *arg);
}
```

What value is printed on a big-endian 64-bit computer?

a) 0  
b) 1  
c) $2^{32}$  
d) $2^{32}-1$
This code exists in `mem.c`. We compile it with the command "gcc –o mem mem.c". We then run it, but the abort function causes it to terminate, producing a core dump – i.e., a file (named core) containing the entire contents of the program’s memory. We examine this file using the command "hexdump –C core", which prints out the contents of the file in hex digits, as well as the ASCII interpretation of those digits. From this, it will become clear that our computers (running Intel x86-64 processors) are little-endian.
Fractional binary numbers

- What is $1011.101_2$?
**Fractional Binary Numbers**

- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: \( \sum_{k=j}^i b_k \times 2^k \)

Supplied by CMU.
Representable Numbers

• Limitation #1
  – can exactly represent only numbers of the form n/2^k
    » other rational numbers have repeating bit representations
  – value representation
    » 1/3 \(0.010101010[01]_{-2}\)
    » 1/5 \(0.001100110011[0011]_{-2}\)
    » 1/10 \(0.0001100110011[0011]_{-2}\)

• Limitation #2
  – just one setting of decimal point within the w bits
    » limited range of numbers (very small values? very large?)
IEEE Floating Point

- **IEEE Standard 754**
  - established in 1985 as uniform standard for floating point arithmetic
    » before that, many idiosyncratic formats
  - supported on all major CPUs

- **Driven by numerical concerns**
  - nice standards for rounding, overflow, underflow
  - hard to make fast in hardware
    » numerical analysts predominated over hardware designers in defining standard

Supplied by CMU.
Floating-Point Representation

- **Numerical Form:**
  \((-1)^s \ M \ \times \ 2^e\)
  - sign bit \(s\) determines whether number is negative or positive
  - significand \(M\) normally a fractional value in range \([1.0, 2.0)\)
  - exponent \(e\) weights value by power of two

- **Encoding**
  - 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 \(M\))
Supplied by CMU.

On x86 hardware, all floating-point arithmetic is done with 80 bits, then reduced to either 32 or 64 as required.
“Normalized” Values

- When: exp ≠ 000...0 and exp ≠ 111...1

- Exponent coded as biased value: E = Exp − Bias
  - exp: unsigned value exp
  - bias = 2^{k-1} - 1, where k is number of exponent bits
    » single precision: 127 (Exp: 1...254, E: -126...127)
    » double precision: 1023 (Exp: 1...2046, E: -1022...1023)

- Significand coded with implied leading 1: M = 1.xxx...x2
  - xxx...x: bits of frac
  - minimum when frac=000...0 (M = 1.0)
  - maximum when frac=111...1 (M = 2.0 − ε)
  - get extra leading bit for “free”
Normalized Encoding Example

- **Value:** \( \text{float } F = 15213.0; \)
  - \( 15213_{10} = 11101101101101_2 \)
  - \( = 1.1101101101101_2 \times 2^{13} \)

- **Significand**
  - \( M = \overline{1.1101101101101}_2 \)
  - \( \text{frac} = \overline{110110111010000000000}_2 \)

- **Exponent**
  - \( E = 13 \)
  - \( bias = 127 \)
  - \( exp = 140 = \overline{1000110}_2 \)

- **Result:**

  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  \hline
  0 & 10001100 & 11011011011010000000000000000000 \\
  \end{array}
  \]

(Supplied by CMU.)
Denormalized Values

- Condition: \( \text{exp} = 000...0 \)
- Exponent value: \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))
- Significand coded with implied leading 0:
  \( M = 0.xxx...x_2 \)
  - \( xxx...x \): bits of \( \text{frac} \)
- Cases
  - \( \text{exp} = 000...0, \text{frac} = 000...0 \)
    - represents zero value
    - note distinct values: +0 and −0 (why?)
  - \( \text{exp} = 000...0, \text{frac} \neq 000...0 \)
    - numbers closest to 0.0
    - equispaced
Special Values

- **Condition: \( \exp = 111...1 \)**

- **Case: \( \exp = 111...1, \frac{\text{c}}{\text{d}} = 000...0 \)**
  - represents value \( \infty \) (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., \( 1.0/0.0 = -1.0/-0.0 = \infty, \ 1.0/-0.0 = -\infty \)

- **Case: \( \exp = 111...1, \frac{\text{c}}{\text{d}} \neq 000...0 \)**
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
Tiny Floating-Point Example

- **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 frac

- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Supplied by CMU.
# Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>00000 001</td>
<td>-6</td>
<td>$1/8*1/64$</td>
<td>$1/512$</td>
</tr>
<tr>
<td>0</td>
<td>00000 010</td>
<td>-6</td>
<td>$2/8*1/64$</td>
<td>$2/512$</td>
</tr>
<tr>
<td>...</td>
<td>0 0000 110</td>
<td>-6</td>
<td>$6/8*1/64$</td>
<td>$6/512$</td>
</tr>
<tr>
<td>...</td>
<td>0 0001 000</td>
<td>-6</td>
<td>$8/8*1/64$</td>
<td>$8/512$</td>
</tr>
<tr>
<td>...</td>
<td>0 0011 000</td>
<td>-6</td>
<td>$9/8*1/64$</td>
<td>$9/512$</td>
</tr>
<tr>
<td>...</td>
<td>0 0110 110</td>
<td>-1</td>
<td>$14/8*1/2$</td>
<td>$14/16$</td>
</tr>
<tr>
<td>...</td>
<td>0 0111 000</td>
<td>0</td>
<td>$8/8*1$</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>0 0111 010</td>
<td>0</td>
<td>$9/8*1$</td>
<td>9/8</td>
</tr>
<tr>
<td>...</td>
<td>0 1110 110</td>
<td>7</td>
<td>$14/8*128$</td>
<td>$224$</td>
</tr>
<tr>
<td>...</td>
<td>0 1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>

- **Denormalized numbers**
- **Normalized numbers**

- **Closest to zero**
- **Largest denorm**
- **Smallest norm**
- **Closest to 1 below**
- **Closest to 1 above**
- **Largest norm**

---

Supplied by CMU.
Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is $2^{3-1} - 1 = 3$

- Notice how the distribution gets denser toward zero.

Supplied by CMU.
Supplied by CMU.

Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3

```
\[ s \quad \text{exp} \quad \text{frac} \]
\[ 1 \quad 3\text{-bits} \quad 2\text{-bits} \]
```

-1  0  0.5  1

- Denormalized
- Normalized
- Infinity
Quiz 2

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3

What number is represented by 0 011 10?

a) 12
b) 1.5
c) .5
d) none of the above
Floating-Point Operations: Basic Idea

- $x +_{\varepsilon} y = \text{Round}(x + y)$
- $x \times_{\varepsilon} y = \text{Round}(x \times y)$

**Basic idea**
- first *compute exact result*
- make it fit into desired precision
  » possibly overflow if exponent too large
  » possibly *round to fit into* $frac$

Supplied by CMU.
# Rounding

- Rounding modes (illustrated with $ 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>towards 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>nearest integer</td>
<td>$1</td>
<td>$2</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>nearest even (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>−$2</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Creating a Floating Point Number

• **Steps**
  – normalize to have leading 1
  – round to fit within fraction
  – postnormalize to deal with effects of rounding

• **Case study**
  – convert 8-bit unsigned numbers to tiny floating-point format

example numbers

<table>
<thead>
<tr>
<th></th>
<th>128</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
<td>00001101</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>00001000</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>00010011</td>
</tr>
<tr>
<td></td>
<td>138</td>
<td>10001010</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>00111111</td>
</tr>
</tbody>
</table>
Normalize

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4-bits</td>
<td>3-bits</td>
</tr>
</tbody>
</table>

- Requirement
  - set binary point so that numbers of form 1.xxxxx
  - adjust all to have leading one
    » decrement exponent as shift left

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>1.0000000</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>1.1010000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.0001000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.0011000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.0001010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</td>
<td>5</td>
</tr>
</tbody>
</table>
Rounding

1. BBGRXXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- Round-up conditions
  - round = 1, sticky = 1 ⇒ > 0.5
  - guard = 1, round = 1, sticky = 0 ⇒ round up to even

<table>
<thead>
<tr>
<th>Value</th>
<th>Fraction</th>
<th>GRS</th>
<th>Incr?</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>13</td>
<td>1.1010000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.0001000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.0011000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111000</td>
<td>111</td>
<td>Y</td>
<td>10.000</td>
</tr>
</tbody>
</table>

Supplied by CMU.
## Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded</th>
<th>Exp</th>
<th>Adjusted</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.000</td>
<td>7</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>13</td>
<td>1.101</td>
<td>3</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>63</td>
<td>10.000</td>
<td>5</td>
<td>1.000*2^6</td>
<td>64</td>
</tr>
</tbody>
</table>
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 \(s\): \(s_1 \oplus s_2\)
  - significand \(M\): \(M_1 \times M_2\)
  - exponent \(E\): \(E_1 + E_2\)

- **Fixing**
  - if \(M \geq 2\), shift \(M\) right, increment \(E\)
  - if \(E\) out of range, overflow (or underflow)
  - round \(M\) to fit \(\text{frac}\) precision

- **Implementation**
  - biggest chore is multiplying significands

Supplied by CMU.

Note that to compute \(E\), one must first convert \(\text{exp}_1\) and \(\text{exp}_2\) to \(E_1\) and \(E_2\), then add them together and check for underflow or overflow (corresponding to \(-\infty\) and \(+\infty\)), and then convert to exp.
Supplied by CMU.

Note that, by default, overflow results in either $+\infty$ or $-\infty$. 
Floating Point in C

• C guarantees two levels
  - float single precision
  - double double precision

• Conversions/casting
  - casting between int, float, and double changes bit representation
  - double/float → int
    » truncates fractional part
    » like rounding toward zero
    » not defined when out of range or NaN: generally sets to Tmin
  - int → double
    » exact conversion, as long as int has ≤ 53-bit word size
  - int → float
    » will round according to rounding mode
Quiz 3

Suppose \( f \), declared to be a float, is assigned the largest possible floating-point positive value (other than \( +\infty \)). What is the value of \( g = f + 1.0 \)?

a) \( f \)
b) \( +\infty \)
c) NAN
d) 0
Float is not Rational ...

- Floating addition
  - commutative: \( a +^f b = b +^f a \)
    » yes!
  - associative: \( a +^f (b +^f c) = (a +^f b) +^f c \)
    » no!
      • \( 2 +^f (1e20 -1e20) = 2 \)
      • \( (2 +^f 1e20) +^f -1e20 = 0 \)

Note that the floating-point numbers in this and the next two slides are expressed in base 10, not base 2.
Float is not Rational …

• Multiplication
  – commutative: \( a \times f b = b \times f a \)
    » yes!
  – associative: \( a \times f (b \times f c) = (a \times f b) \times f c \)
    » no!
    • \( 1e20 \times f (1e20 \times f 1e-20) = 1e20 \)
    • \( (1e20 \times f 1e20) \times f 1e-20 = +\infty \)
Float is not Rational …

- More …
  - multiplication distributes over addition:
    \[ a^{\text{f}} (b^{\text{f}} + c) = (a^{\text{f}} b) +^{\text{f}} (a^{\text{f}} c) \]
    » no!
    » \(1e20^{\text{f}} (1e20^{\text{f}} -1e20) = 0\)
    » \((1e20^{\text{f}} 1e20) +^{\text{f}} (1e20^{\text{f}} -1e20) = \text{NaN}\)
  - loss of significance:
    \[ x=y+1 \]
    \[ z=2/(x-y) \]
    \[ z==2? \]
    » not necessarily!
    - consider \(y = 1e20\)