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)

Supplied by CMU.
Byte Ordering

- Four-byte integer
  - 0x76543210
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

<table>
<thead>
<tr>
<th>76</th>
<th>54</th>
<th>32</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>32</th>
<th>54</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Little-endian
Big-endian

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
    - \(1/3\) \(0.0101010101[01]_2\)
    - \(1/5\) \(0.001100110011[0011]_2\)
    - \(1/10\) \(0.0001100110011[0011]_2\)

- Limitation #2
  - just one setting of decimal point within the \(w\) bits
    - limited range of numbers (very small values? very large?)

Supplied by CMU.
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
Floating-Point Representation

- Numerical Form:
  \((-1)^s \times M \times 2^e\)
  - 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: \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)

- Exponent coded as biased value: \( E = \text{Exp} - \text{Bias} \)
  - \( \text{exp} \): unsigned value \( \text{exp} \)
  - \( \text{bias} = 2^{k-1} - 1 \), 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: \( M = 1.\text{xxx...x}_2 \)
  - \( \text{xxx...x} \): bits of \( \text{frac} \)
  - minimum when \( \text{frac}=000\ldots0 \) (M = 1.0)
  - maximum when \( \text{frac}=111\ldots1 \) (M = 2.0 – \( \epsilon \))
  - get extra leading bit for “free”
Normalized Encoding Example

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

- **Significand**
  - $M = 1.1101101101101_2$
  - $\text{frac} = 110110110110100000000000002$

- **Exponent**
  - $E = 13$
  - $bias = 127$
  - $exp = 140 = 10001100_2$

- **Result:**

```
  U 10001100 11011011011010000000000000
  s  exp  frac
```

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

Supplied by CMU.
Special Values

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

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

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

Supplied by CMU.
Tiny Floating-Point Example

<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>

- **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>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>(1/8 \times 1/64 = 1/512) closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>(2/8 \times 1/64 = 2/512)</td>
</tr>
</tbody>
</table>

**Denormalized numbers**

<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>0000</td>
<td>110</td>
<td>-6</td>
<td>(6/8 \times 1/64 = 6/512) largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>(7/8 \times 1/64 = 7/512)</td>
</tr>
</tbody>
</table>

<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>0001</td>
<td>000</td>
<td>-6</td>
<td>(8/8 \times 1/64 = 8/512) smallest norm</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>(9/8 \times 1/64 = 9/512)</td>
</tr>
</tbody>
</table>

**Normalized numbers**

<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>0110</td>
<td>110</td>
<td>-1</td>
<td>(14/8 \times 1/2 = 14/16) closest to 1 below</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>(15/8 \times 1/2 = 15/16)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>(8/8 \times 1 = 1) closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>(9/8 \times 1 = 9/8)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>(10/8 \times 1 = 10/8)</td>
</tr>
</tbody>
</table>

<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>1110</td>
<td>110</td>
<td>7</td>
<td>(14/8 \times 128 = 224) largest norm</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>(15/8 \times 128 = 240)</td>
</tr>
</tbody>
</table>

<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>1111</td>
<td>000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

---

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.
Distribution of Values (close-up view)

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

Supplied by CMU.
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 \( \text{frac} \)

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>
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**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td></td>
</tr>
</tbody>
</table>
Normalize

<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>1.00000000</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>1.10100000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.00010000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.00110000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.00010100</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</td>
<td>5</td>
</tr>
</tbody>
</table>
Rounding

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>1.00000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>13</td>
<td>1.10100000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.00010000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.00110000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.00010100</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.11111000</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: \(s_1 \land s_2\)
  - 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 frac precision

- Implementation
  - biggest chore is multiplying significands

Supplied by CMU.

Note that to compute E, one must first convert \(\exp_1\) and \(\exp_2\) 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.
Floating-Point Addition

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

- **Exact result: $(\pm 1)^{s} M \ 2^{E}$**
  - sign $s$, significand $M$:
    » result of signed align & add
  - exponent $E$: $E_1$

- **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

Supplied by CMU.

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