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.
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - \( u \gg k \) gives \( \lfloor \frac{u}{2^k} \rfloor \)
  - uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>76 66</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>95 03</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59 00</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - uses arithmetic shift
  - rounds wrong direction when \( x < 0 \)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y \gg 1</td>
<td>-7606.5</td>
<td>E2</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y \gg 4</td>
<td>-950.8125</td>
<td>FC</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y \gg 8</td>
<td>-59.4257813</td>
<td>FF</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>

Supplied by CMU.
**Correct Power-of-2 Divide**

- Quotient of negative number by power of 2
  - want ⌊x / 2^k⌋ (round toward 0)
  - compute as ⌊(x+2^{k-1}) / 2^k⌋
    - in C: (x + (1<<k)-1) >> k
    - biases dividend toward 0

**Case 1: no rounding**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>+2^k-1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>divisor:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>binary point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>⌊u / 2^k⌋</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Biases has no effect*
Correct Power-of-2 Divide (Cont.)

Case 2: rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( x + 2^k - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( \cdots ) 1 ( \cdots ) 1 ( k )</td>
<td>0 ( \cdots ) 001 ( \cdots ) 11</td>
</tr>
</tbody>
</table>

incremented by 1

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( \frac{x}{2^k} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( \cdots ) 111 ( \cdots ) 1</td>
<td>0 ( \cdots ) 010 ( \cdots ) 0</td>
</tr>
</tbody>
</table>

Biased adds 1 to final result
Why Should I Use Unsigned?

- Don’t use just because number nonnegative
  - easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  - can be very subtle
    ```c
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
        ...
    ```
- Do use when using bits to represent sets
  - logical right shift, no sign extension

Supplied by CMU.

Note that “sizeof” returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)
Combining Bytes

- Data items of multiple sizes, usually powers of two
  - one-byte, two-byte, four-byte, eight-byte integers
  - four-byte and eight-byte floating-point numbers
- For example: four consecutive bytes interpreted as storing an integer (or a float)
  - for best performance, address of lowest byte should be a multiple of the size of the item (four in this case)

The reason for the performance issue has to do with how the memory subsystem works, somethings that will be explained in a few weeks.
Word Size

- (Mostly) obsolete term
  - old computers had items of one size: the word size
- Now used to express the number of bits necessary to hold an address
  - 16 bits (really old computers)
  - 32 bits (old computers)
  - 64 bits (most current computers)
Byte Ordering

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

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

Little-endian

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

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 prints out the value of x, one byte at a time, starting with the byte at the lowest address (little end). On x86-based computers, it will print:

00010203

which means that the address of an int is the address of the byte containing its least significant digits (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: \[ \sum_{k=-j}^{i} b_k \times 2^k \]
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?)
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:**
  \((-1)^s M \times 2^e\)
  - 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 \(M\))

```
  s   exp   frac
```
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: $\exp \neq 000...0$ and $\exp \neq 111...1$

- 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: $M = 1.xxx...x_2$
  - $xxx...x$: bits of $\frac{1}{2}$
  - minimum when $\frac{1}{2} = 000...0$ ($M = 1.0$)
  - maximum when $\frac{1}{2} = 111...1$ ($M = 2.0 - \epsilon$)
  - get extra leading bit for “free”
Normalized Encoding Example

- **Value:** float $F = 15213.0$;
  - $15213_{10} = 1110110111011_{2}$
  - $= 1.110110111011_{2} \times 2^{13}$

- **Significand**
  - $M = 1.110110111011_{2}$
  - $frac = 11011011101000000000000_{2}$

- **Exponent**
  - $E = 13$
  - $bias = 127$
  - $exp = 140 = 10001100_{2}$

- **Result:**
  - $s \quad exp \quad frac$

---

Supplied by CMU.
Denormalized Values

• Condition: \( \text{exp} = 000\ldots0 \)
• Exponent value: \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))
• Significand coded with implied leading 0: 
  \[ M = 0.xxx\ldots x_2 \]
  – \( xxx\ldots x \): bits of \( \text{frac} \)
• Cases
  – \( \text{exp} = 000\ldots0, \text{frac} = 000\ldots0 \)
    » represents zero value
    » note distinct values: \(+0\) and \(-0\) (why?)
  – \( \text{exp} = 000\ldots0, \text{frac} \neq 000\ldots0 \)
    » 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

- 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
## 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 * 1/64 = 1/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>2/8 * 1/64 = 2/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest denorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>smallest norm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest norm</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 * 1/64 = 6/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>7/8 * 1/64 = 7/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest denorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>smallest norm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest norm</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 * 1/2 = 14/16</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>15/8 * 1/2 = 15/16</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8 * 1 = 1</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8 * 1 = 9/8</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8 * 1 = 10/8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest norm</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 * 128 = 224</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8 * 128 = 240</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>inf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest norm</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** frac
Rounding

- Rounding modes (illustrated with $ rounding)

<table>
<thead>
<tr>
<th>Method</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 (-\infty)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$-2</td>
</tr>
<tr>
<td>round up (+\infty)</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>Decimal</th>
<th>Binary 8-bit</th>
<th>10-bit Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>100000000000111</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>000011010000000</td>
</tr>
<tr>
<td>33</td>
<td>00010011</td>
<td>000100110000000</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
<td>000100110000000</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>010001010000000</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>001111110000000</td>
</tr>
</tbody>
</table>
Normalize

- 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

Guard bit: LSB of result

Round bit: 1\textsuperscript{st} bit removed

Sticky bit: OR of remaining bits

- Round-up conditions
  - \( \text{round} = 1, \text{sticky} = 1 \Rightarrow > 0.5 \)
  - \( \text{guard} = 1, \text{round} = 1, \text{sticky} = 0 \Rightarrow \text{round up to even} \)

<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>1.1111100</td>
<td>111</td>
<td>Y</td>
<td>1.000</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

- \((-1)^{s_1} M_1 \times 2^{E_1} \times (-1)^{s_2} M_2 \times 2^{E_2}\)
- **Exact result:** \((-1)^s M \times 2^E\)
  - sign \(s\): \(s_1 \And s_2\)
  - 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 \(\text{float}\) 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 \(\exp\).
Supplied by CMU.

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

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: $(1)^{s} M 2^{e}$
  - 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

CS33 Intro to Computer Systems