Floating Point ranges

- While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it.
- Values of all 0s in this field are reserved for the zeros and subnormal numbers, values of all 1s are reserved for the infinities and NaNs.
- The exponent range for normalized numbers is \([-126, 127]\) for single precision and \([-1022, 1023]\) for double.
- Normalized numbers exclude subnormal values, zeros, infinities, and NaNs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Sign</th>
<th>Exponent</th>
<th>Significand</th>
<th>Total bits</th>
<th>Exponent bias</th>
<th>Bits precision</th>
<th>#decimal digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half (IEEE 754-2008)</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>~3.3</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>8</td>
<td>23</td>
<td>32</td>
<td>127</td>
<td>24</td>
<td>~7.2</td>
</tr>
<tr>
<td>Double</td>
<td>1</td>
<td>11</td>
<td>52</td>
<td>64</td>
<td>1023</td>
<td>53</td>
<td>~15.9</td>
</tr>
</tbody>
</table>
REMINDER: floating point arithmetic can only approximate real arithmetic, since the representation has limited range and precision
IEEE supported
- Round to even (nearest) – default mode
- Round toward zero (for integer truncation)
- Round down
- Round up

What happens when the value is halfway between two possibilities???
- ROUND TO EVEN → rndtest.c
Casting

- From int to float
  - The number cannot overflow but may be rounded

- From int or float to double
  - The exact numeric value can be preserved because double has both a greater range as well as a greater precision

- From double to float
  - The value can overflow to + or – infinity, since range is smaller
  - Otherwise, maybe be rounded, because the precision is smaller.

- From float or double to int
  - The value will be rounded toward zero
  - The value may overflow (etc)
Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \div \pm\text{Infinity}$</td>
<td>0</td>
</tr>
<tr>
<td>$\pm\text{Infinity} \times \pm\text{Infinity}$</td>
<td>$\pm\text{Infinity}$</td>
</tr>
<tr>
<td>$\pm\text{nonzero} \div 0$</td>
<td>$\pm\text{Infinity}$</td>
</tr>
<tr>
<td>$\text{Infinity} + \text{Infinity}$</td>
<td>$\text{Infinity}$</td>
</tr>
<tr>
<td>$\pm 0 \div \pm 0$</td>
<td>NaN</td>
</tr>
<tr>
<td>$\text{Infinity} - \text{Infinity}$</td>
<td>NaN</td>
</tr>
<tr>
<td>$\pm\text{Infinity} \div \pm\text{Infinity}$</td>
<td>NaN</td>
</tr>
<tr>
<td>$\pm\text{Infinity} \times 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>
.h file MACROS/constants

- Math.h
  - HUGE
  - HUGE_VAL

- Values.h
  - MAXFLOAT
  - MAXINT
  - MAXDOUBLE
  - MAXLONG
  - DMINEXP

- Limits.h
  - [Link to Limits.h](http://en.wikibooks.org/wiki/C_Programming/C_Reference/limits.h)
And so much more!

- Operations
- Exceptions
  - Inexact
  - Overflow
  - Underflow
  - Divide by zero
  - Indefinite (NaN)
- Interchange formats
- There are whole courses just on floating point
CHAPTER 3.2 Program Encodings

- **GOAL** ➔ examine assembly code and map it back to the constructs found in high-level programming languages

- %gcc –O1 –S code.c ➔ code.s
- %more code.s
  - Runs the compiler only
  - -S options = generates an assembly (.s) file
  - -O1 is an optimization level
  - All information about local variables names or data types have been stripped away
  - Still see global variable “accum”
    - Compiler has not yet determined where in memory this variable will be stored

- %gcc –O1 –c –m32 code.c ➔ code.o
- %objdump –d code.o
  - -c compiles and assembles the code
  - Generates an object-code file (.o) = binary format
  - DISASSEMBLER – re-engineers the object code back into assembly language
  - %uname –p
    - Do we have a 32 or 64 bit machine? ➔ X86_64 i.e. 64 bit machine
    - -m32 is a gcc option to run/build 32-bit applications on a 64-bit machine

- Compare code.s from gcc versus code.o from objdump