FLOATING POINT REPRESENTATION
ABOUT FLOATING POINTS

Integer Data Type
- 32-bit unsigned integers limited to whole numbers from 0 to just over 4 billion
  - What about national debt, Avogadro’s number, Googol...the number?
  - 64-bit unsigned integers up to over 9 quintillion
    - What about small numbers and fractions (e.g. 1/2 or $\pi$)?

Requires a different interpretation of the bits!
- Data types in C
  - float (32-bit IEEE floating point format)
  - double (64-bit IEEE floating point format)

- 32-bit int and float both represent $2^{32}$ distinct values!
  - Trade-off range and precision
  - e.g. to support large numbers (> $2^{32}$) and fractions, float can not represent every integer between 0 and $2^{32}$!
FRACTIONAL BINARY NUMBERS

In Base 10, a decimal point for representing non-integer values

- 125.35
  - $1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 + 3 \times 10^{-1} + 5 \times 10^{-2}$

In Base 2, a binary point:

- $b_n b_{n-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-m}$

- 101.11$_2$
  - $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$
  - $4 + 0 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$

Accuracy is a problem

- Numbers such as 1/5 or 1/3 must be approximated
- But this is true also with decimal
PARTNER ACTIVITY

Convert the following binary numbers to decimal mixed numbers:

▸ 10.111\textsubscript{2}
  \[2 + 0 + 1/2 + 1/4 + 1/8 = 2 \ 7/8\]

▸ 1.0111\textsubscript{2}
  \[1 + 1/4 + 1/8 + 1/16 = 1 \ 7/16\]

▸ 1011.101\textsubscript{2}
  \[8 + 2 + 1 + 1/2 + 1/8 = 11 \ 5/8\]
How can we represent very large or very small numbers with a compact representation?

- Current way with int
  - $5 \times 2^{100}$ as 1010000....000000000000? (103 bits)
  - Not very compact, but can represent all integers in between

- Another way...
  - $5 \times 2^{100}$ as 101 01100100 (i.e. $x=101$ and $y=01100100$)? (11 bits)
  - Compact, but does not represent all integers in between

Basis for IEEE Standard 754, “IEEE Floating Point”

- Supported in most modern CPUs via floating-point unit
- Encodes rational numbers in the form $(M \times 2^E)$
  - Large numbers have positive exponent $E$
  - Small numbers have negative exponent $E$
  - Rounding can lead to errors
IEEE STANDARD 754
“IEEE FLOATING POINT”

Specifically, “IEEE Floating Point” represents numbers in the form

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

Three fields
- **S** “sign” bit
  - 1 == Negative
  - 0 == Positive
- **M** “mantissa”, the significand, a fractional number
- **E** “exponent”, could be negative
IEEE FLOATING POINT ENCODING

- **s** is the “sign” bit
  - 1 == Negative
  - 0 == Positive

- **exp** field is an encoding to derive \( E \)
- **frac** field is an encoding to derive \( M \)

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

- **S** “sign” bit
- **M** “mantissa”
- **E** “exponent” (could be negative)
IEEE FLOATING POINT ENCODING

\[ \begin{array}{c c c}
S & \text{exp} & \text{frac} \\
\hline
\text{is the “sign” bit} & \text{field is an encoding to derive } E \\
1 == \text{Negative} & \text{field is an encoding to derive } M \\
0 == \text{Positive} & \\
\end{array} \]

Field Sizes

- **Single** precision, 32-bit encoding ("float" type):
  - 8 exp bits
  - 23 frac bits
- **Double** precision, 64-bit encoding ("double" type):
  - 11 exp bits
  - 52 frac bits
- **Extended** precision, 80-bit encoding (found in Intel FPUs)
  - 15 exp bits + 63 frac bits (1 bit unused)
IEEE FLOATING POINT ENCODING

The exp value is stored in two’s complement form, with a bias. That is, the exp value has a bias added to it.

For “normalized” numbers, the biases are:

For 32-bit (“float”): Bias is 127
For 64-bit (“double”): Bias is 1023

For normalized values, the exp value is never all 0’s (“0”) or all 1’s (“255”). These are reserved for other types of numbers.

Note that this means “0” cannot be represented as a normalized value!
Smooth transition to evenly spaced increments approaching 0
- Allows for very small numbers to be represented
- Allows 0 to be represented
Normalized Numbers: \( \pm 1.B \times 2^E \)

-1.0 \times 2^{-126} \quad 0 \quad +1.0 \times 2^{-126}
De-normalized Numbers: \( \pm 0.B \times 2^E \)

-1.0 \times 2^{-126} \quad 0 \quad +1.0 \times 2^{-126}

De-Normalized
NORMALIZED VS DE-NORMALIZED

32-bit de-normalized Numbers: ±0.B x 2^{-126}

Largest positive de-normalized number:
0.11111111111111111111111 x 2^{-126}

-1.0 x 2^{-126} 0 +1.0 x 2^{-126}

De-Normalized
NORMALIZED VS DE-NORMALIZED

32-bit de-normalized Numbers: $\pm 0.B \times 2^{-126}$

Smallest negative de-normalized number:
$-0.11111111111111111111111\times 2^{-126}$

-1.0 $\times 2^{-126}$ 0 +1.0 $\times 2^{-126}$
Depending on the exp value, the bits are interpreted differently

- **Normalized (most numbers – in the form \( \pm 1.B \times 2^E \))**:
  - **exp** is neither all 0’s nor all 1’s
  - **exp = E + bias**
    - **exp** is in biased form:
      - 127 for single precision (8-bit exp = \( 2^7 - 1 \))
      - 1023 for double precision (11-bit exp = \( 2^{10} - 1 \))
    - Allows for negative exponents
  - **M is 1 + frac**
Depending on the exp value, the bits are interpreted differently

- Denormalized (very small numbers / close to 0 – ±0.B x 2^E):
  - exp is all 0’s
  - E is 1-Bias
    - Not set to –Bias in order to ensure smooth transition from Normalized
  - M is frac
    - Can represent 0 exactly
    - Evenly spaced increments approaching 0
Depending on the exp value, the bits are interpreted differently

- **Denormalized (very small numbers / close to 0 – ±0.B x 2^E):**

This is allows for the following representation of “0”:

0 000000 00000000000000000000000

This is also allows for the following representation of “-0”:

1 000000 00000000000000000000000

The floating point hardware will treat these two values as equal.
Depending on the exp value, the bits are interpreted differently

- **Special values:**
  - If the exp is all 1's
  - If frac == 0, then we have ±∞
    - These are the results of calculations where the positive range of the exponent is exceeded, or division of a regular number by zero.
Depending on the exp value, the bits are interpreted differently

- **Special values:**
  - If the exp is all 1’s

- If frac != 0, we have NaN (Not a Number)
  - There are special not a number (or NaN) values where the exponent is all 1-bits and the significand is not all 0-bits.
  - These represent the result of various undefined calculations (like multiplying 0 and infinity, any calculation involving a NaN value, or application-specific cases).
  - Even bit-identical NaN values must not be considered equal.
“Float Toy”
http://evanw.github.io/float-toy/

“IEEE-754 Visualization”
https://bartaz.github.io/ieee754-visualization/

“Float Exposed”
https://float.exposed/0x40490fdb
NORMALIZED ENCODING EXAMPLE

(Using 32-bit float)

Value

- \( \text{float } f = 15213.0; \) /* exp=8 bits, frac=23 bits */
- \( 15213_{10} = 11101101101101_2 \\
  = 1 \ 1101101101101_2 \\
  = 1.1101101101101_2 \times 2^{13} \) (normalized form)

Significand

- \( M = 1.1101101101101_2 \)
- \( \text{Frac=} \ 1101101101101101101101101101101_2 \\

Exponent \ (\text{Bias} + E = \text{Exp})

- \( E = 13 \)
- \( \text{Bias} = 127 \) (Single Precision!)
- \( \text{Exp} = 140 = 10001100_2 \)
NORMALIZED ENCODING EXAMPLE

Significand
- $M = 1.1101101101101_2$
- $\text{Frac} = 110110110110100000000000000_2$

Exponent (Bias + $E = \text{Exp}$)
- $E = 13$
- Bias = 127 (Single Precision!)
- Exp = 140 = $10001100_2$

Floating Point Representation:
Sign: 0
140: 100 0110 0
15213: 110 1101 1011 0100 0000 0000

Binary: 0100 0110 0110 1101 1011 0100 0000 0000
Hex: 4 6 6 D B 4 0 0
FLOATING POINT OPERATIONS

Floating Point Addition

- Commutative
  - \( x + y = y + x \)
- NOT associative:
  - \((x + y) + z \neq x + (y + z)\)
    - \((3.14 + 1010) - 1010 = 0.0\) (due to rounding)
    - \(3.14 + (1010 - 1010) = 3.14\)
- Very important for scientific and compiler programmers

Floating Point Multiplication

- Is not associative
- Does not distribute over addition
  - \(10^{20} \times (10^{20} - 10^{20}) = 0.0\)
  - \(10^{20} \times 10^{20} - 10^{20} \times 10^{20} = \text{NaN}\)
- Again, very important for scientific and compiler programmers
FLOATING POINT IN C

C guarantees two levels

- float single precision
- double double precision

Casting between data types (not pointer types)

- Casting between int, float, and double results in (sometimes inexact) conversions to the new representation
- **float to int**
  - Not defined when beyond range of int
  - Generally saturates to $T_{\text{Min}}$ or $T_{\text{Max}}$
- **double to int**
  - Same as with float
- **int to double**
  - Exact conversion
- **int to float**
  - Will round for large values (e.g. that require > 23 bits)
int x
x == (int)(float) x

Compiled with `gcc -O2`, this is true!
(For example, with `x = 2147483647`)

What's going on?

- See Computer Systems Book, Ch. 2.4.6
- Two potential optimizations
  - x86 use of 80-bit floating point registers
  - Compiler skips useless cast
- Non-optimized code returns results into memory
  - 32 bits for intermediate float
Microsoft Calculator: The sqrt (square root) estimator

sqrt(4) - 2 =
INFAMOUS ERRORS

Ariane 5 Rocket

- Around 40 seconds into launch, the rocket’s computers decided it was 90 degrees off course, and “corrected” itself.
- Caused by floating point cast to integer for efficiency, ended up in overflow trap.
- $7 billion dollars in R&D, Cargo valued at $500 million
INFAMOUS ERRORS

Patriot Missile

- Rounding error from inaccurate representation of 1/10 in time calculations
- 28 killed due to failure in intercepting Scud missile (2/25/1991)
Patriot Missile

Specifically, the time in tenths of second as measured by the system’s internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, led to a significant error.

Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds.

A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time.
In the book, Problem 2.47

Given that (assume neither d nor f is NaN):

\[
\text{int } x; \quad \text{float } f; \quad \text{double } d;
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = (\text{int})(\text{float}) \ x )</td>
<td>No: 23 bit frac</td>
</tr>
<tr>
<td>( x = (\text{int})(\text{double}) \ x )</td>
<td>Yes: 52 bit frac</td>
</tr>
<tr>
<td>( f = (\text{float})(\text{double}) \ f )</td>
<td>Yes: Increases precision</td>
</tr>
<tr>
<td>( d = (\text{float}) \ d )</td>
<td>No: Loses precision</td>
</tr>
<tr>
<td>( f = -(-f) )</td>
<td>Yes: Just change sign bit</td>
</tr>
<tr>
<td>( 2/3 = 2/3.0 )</td>
<td>No: ( 2/3 = 0 )</td>
</tr>
<tr>
<td>( d &lt; 0.0 ((d*2) &lt; 0.0) )</td>
<td>Yes: (Note use of -)</td>
</tr>
<tr>
<td>( d &gt; f ) (-f &gt; -d)</td>
<td>Yes</td>
</tr>
<tr>
<td>( d \times d &gt;= 0.0 )</td>
<td>Yes: (Note use of +)</td>
</tr>
<tr>
<td>((d+f)-d = f)</td>
<td>No: Not associative</td>
</tr>
</tbody>
</table>