15213 Recitation 2: Floating Point

1 Introduction

This handout will introduce and test your knowledge of the floating point representation of real numbers, as defined by the IEEE standard. This information may be useful for datalab.

2 Scientific Notation

1. Write 15213 in (decimal) scientific notation. Describe the different parts of the notation.

   Answer: $15213 = 1.5213 \cdot 10^4$. There’s the fractional part and the exponent.

2. Write out the numbers in table 1 using (binary) scientific notation. In this notation, we use powers of 2 rather than powers of 10.

3. What values are to the left of the binary point in each number?

   Answer: 0111

\[\text{Table 1: Complete the table below.}\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Scientific Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 \frac{3}{4}$</td>
<td>101.11</td>
<td>$1.0111 \cdot 2^2$</td>
</tr>
<tr>
<td>$2 \frac{7}{8}$</td>
<td>10.111</td>
<td>$1.0111 \cdot 2^1$</td>
</tr>
<tr>
<td>$1 \frac{7}{16}$</td>
<td>1.0111</td>
<td>$1.0111 \cdot 2^0$</td>
</tr>
</tbody>
</table>
3 IEEE Representation

This section starts the review of the IEEE floating point representation. A single precision float refers to a standardized floating point format using 32 bits, and is the format we use in datalab. Single precision floating point values are split up as follows:

\[
\begin{array}{c}
\text{sign} \quad xxxxxxxx \quad \text{exponent} \quad xxx \cdots xxx \\
\end{array}
\]

IEEE floats are composed of three parts:

1. the sign—always 1 bit;

2. the exponent—8 bits for single-precision floats;

3. the mantissa—23 bits for single-precision floats.

The sign is 1 if the value is negative, and 0 if the value is positive. In normalized form—this will be discussed later in the handout—the actual exponent is equal to the bit value minus a bias value. The bias is calculated by \(2^{(\text{# exponent bits})−1} − 1\). The mantissa is read with an implied leading ‘1.’ at the beginning, so for example a mantissa of 101 represents a value of 1.101. Then it all goes together in scientific notation.

Calculation: \(\text{Value} = (-1)^s \cdot 1.f \cdot 2^E\), where \(s\) is the value of the sign bit, \(f\) is bit representation of the mantissa, and \(E = \text{exponent} − \text{bias}\). Careful! This only holds for normalized values—other kinds will be introduced later in this handout.

1. What is the bias for a 32-bit float? What about a 64-bit float?
   For 32 bits, the bias is \(2^{8−1} − 1 = 127\), for 64 bits it’s \(2^{11−1} − 1 = 1023\).

2. The fractional bits, \(f\), have an implicit 1 in the front. Returning to table[1], what are the values for \(f\) in each number?
   \[
   \begin{array}{l}
   \frac{5}{4}: \quad 0111 \\
   \frac{7}{8}: \quad 0111 \\
   \frac{1}{16}: \quad 0111 \\
   \end{array}
   \]

3. With the 8 exponent bits of single-precision float, what is the smallest non-zero number with 0x01 as the exponent bits? Is this value greater or less than one? Express your answer as a power of 2.
   \[
   1.0000 \cdot 2^{1−127} = 2^{-126}.\]

\(^{23\text{ bits long}}\)
4 Extreme Exponents

1. For what you know of the representation (i.e. normalized) so far, can you represent 0 with the IEEE format?

One of the goals of the IEEE representation was to make the bit pattern of 0 be the value of 0. Therefore, in the IEEE representation, the exponent bit pattern being 0 is a special case. If the exponent is all 0 then the exponent value is set to be $1 - \text{bias}$. Furthermore, the fractional part is denormalized, such that it no longer has a leading 1, i.e. the exponent value is $0.f$.

2. How many representations for zero are there with denormalized floats?

Two. They are positive and negative 0, i.e. 0x0 and 0x80000000.

3. For a single-precision floating point number, using your new knowledge about denormalized form, what is the smallest positive (> 0) number that you can represent?

Give your answer as a power of 2.

$$0.00000000000000000000001 \cdot 2^{1-127} = 2^{-149}.$$ 

One other special case: when the bit representation has all 1s in the exponent places, then the value is either NaN—not a number—or infinity. If the mantissa is 0 then this value is infinity, otherwise it is NaN.

It is standard when dividing by 0 to return either plus or minus infinity, and when performing any operation involving NaN the result will remain NaN. NaN can also be generated by certain expressions which have no other reasonable real value to return, such as $0.0 / 0$ or log($-1$).

4. Give the hex representation of the value produced by $1.0 / 0$.

This produces positive infinity, which is 0x7F800000.

5 Addition

To perform addition with floating point numbers, the hardware first works to line up the binary points. And then it can add the significands together. Finally, the result must be renormalized and a new exponent computed.

1. Compute the sum of the following binary numbers: $1.010 \cdot 2^2 + 1.110 \cdot 2^3$. Apply each step in turn.

We first shift the decimal point to add $1.010 \cdot 2^2 + 11.10 \cdot 2^2 = 100.110 \cdot 2^2 = 1.0011 \cdot 2^4$. 

\footnote{In datalab, when a float function is given NaN it is important that you return the same bit pattern that you have been given, rather than some other bit pattern which also represents NaN.}
Table 2: Fill in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 5/16</td>
<td>1.0101</td>
<td>1.01</td>
<td>1 1/4</td>
</tr>
<tr>
<td>1 3/32</td>
<td>1.00011</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>1 5/32</td>
<td>1.00101</td>
<td>1.01</td>
<td>1 1/4</td>
</tr>
<tr>
<td>1 7/8</td>
<td>1.111</td>
<td>10.0</td>
<td>2</td>
</tr>
<tr>
<td>1 5/8</td>
<td>1.101</td>
<td>1.10</td>
<td>1 1/2</td>
</tr>
</tbody>
</table>

2. How many fractional bits were required for the result of the previous question?
   Four, because the first leading 1 is implied.

3. There are several possible rounding schemes for floating point values. There are two components of rounding. First, is what to do in general? Should the float be rounded up, down, to zero, or to nearest? The second component is what to do about ties with round to nearest. So should 9 / 2.0 be 4 or 5? The default IEEE scheme is to round to nearest even. Apply it to the following values in table 2 for a system that only has three bits for the final values.

6 Simple Floating Point

Throughout the following section, we are going to use a smaller representation: 1 bit for the sign; 3 bits for the exponent, with a bias of 3; and 4 fractional bits.

1. What is the largest absolute value that we can represent with these bits? Smallest non-zero?
   Largest: $1.1111 \cdot 2^{6-3} = 15 \frac{1}{2}$. Smallest positive: $0.0001 \cdot 2^{1-3} = \frac{1}{64}$.

2. Given the following value 0x5C, what is this 8-bit float’s decimal value? What is the binary representation of this value?
   $0x5C = 0101 1100$ so $1.1100 \cdot 2^{5-3} = 7$. In binary, 111.

3. Compute the sum of the following floats (hex values shown): 0x5C + 0x43.
   We add them as we did before, $1.11 \cdot 2^2 + 1.0011 \cdot 2^1 = 11.1 \cdot 2^1 + 1.0011 \cdot 2^1 = 100.1011 \cdot 2^1 = 1.001011 \cdot 2^3$, then we must round to 1.0011 \cdot 2^3, which is 0 110 0011 or 0x63.

4. Compute the product of the following floats (hex values shown): 0x5C \* 0x43. Hint: do multiplication the same way you did in grade school.

---

4 Bonus question: think about why this scheme makes the most sense.
1.11 \cdot 2^2 \cdot 1.0011 \cdot 2^1 = 10.000101 \cdot 2^3 = 1.0000101 \cdot 2^4 \text{ which we round to } 1.0001 \cdot 2^4.

But we’ve run into a problem! 4 is a larger exponent than we can represent, as 4 plus the bias is 7, which means we would need an exponent of all 1s, yet that is a special case as noted above. Therefore we experience overflow, and simply return positive infinity, i.e. 0 111 0000 or 0x70.

7 Bonus Puzzles

7.1 Bit Puzzle

Given a 32-bit float, convert the float to an integer assuming that the value is within the range (TMIN, TMAX) using integer bitwise and shift operations.

unsigned float2i(unsigned);

1. Start by identifying the fields of the float. Write code to assign each to a separate variable.

2. Convert each field from its stored form to its full value.

3. Apply the exponent to the fractional bits.

4. What other components exist in the float?

unsigned float2i(unsigned x)
{
    unsigned const bias = 127;
    unsigned sign = (x >> 31) & 1;
    unsigned exponent_bits = (x >> 23) & 0xFF;
    unsigned mantissa = x & 0x7FFFFF;
    unsigned exponent_value, mantissa_value;
    int shift_amount, value;
    if (exponent_bits) {
        /*
         * It would be good to check for infinity or NaN here, but we
         * are promised it won’t happen.
         */
        exponent_value = exponent_bits – bias;
        mantissa_value = mantissa | 0x800000;
    } else {
        exponent_value = 1 – bias;
        mantissa_value = mantissa;
    }
}
We subtract 23 because if mantissa value were divided by $2^{23}$ it would be the correct value.

```c
shift_amount = exponent_value - 23;
if (shift_amount >= 0) {
  value = mantissa_value << shift_amount;
} else {
  unsigned shift_pos = -shift_amount;
  unsigned guard_bit = (mantissa_value >> shift_pos) & 1;
  unsigned round_bit = (mantissa_value >> (shift_pos - 1)) & 1;
  unsigned sticky_bits = mantissa_value & ((1 << (shift_pos - 1)) - 1);
  int round;
  if (round_bit && (sticky_bits || guard_bit))
    round = 1;
  else
    round = 0;
  value = (mantissa_value >> (-shift_amount)) + round;
}
return sign ? -value : value;
```

7.2 Float?

1. Will the following code ever terminate? If so, what value will the sum contain?

```c
float sum = 0.0f;
float inc = 1.0f;
float tsum;

do {
  tsum = sum;
  sum += inc;
} while (tsum != sum);
```

The code will in fact terminate. It will terminate exactly when tsum becomes infinity, because when sum is incremented enough times, it will eventually hit infinity, at which point adding one does not change the value.

2. If the floats and constants are changed to ints, how (if at all) do your answers change?

Eventually sum will overflow, causing undefined behavior. If we assume twos complement representation, then the code will never terminate, because sum will continue
to wrap around and tsum will always either be sum – 1 or will be TMAX and sum will be TMIN.