Floating point Representation of Numbers

Floating Point representation is useful for representing a number in a wide range: very small to very large. It is widely used in the scientific world. Consider, the following FP representation of a number:

Exponent E  significand F (also called mantissa)

```
+/-  x  x  x  x  y  y  y  y  y  y  y  y  y  y  y  y
```

In **decimal** it means (+/-) 1. yyyyyyyyyyyyy x 10^xxx
In **binary**, it means (+/-) 1. yyyyyyyyyyyyy x 2^xxx
(The 1 is implied)
IEEE 754 single-precision (32 bits)

<table>
<thead>
<tr>
<th>s</th>
<th>xxxxxxxx</th>
<th>yyyyyyyyyyyyyyyyyyyyyyyy</th>
<th>Single precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>23 bits</td>
<td></td>
</tr>
</tbody>
</table>

Largest = $1.111\ldots \times 2^{+127} \approx 2 \times 10^{+38}$

Smallest = $1.000\ldots \times 2^{-128} \approx 1 \times 10^{-38}$

These can be positive and negative, depending on $s$.

(But there are exceptions too)

IEEE 754 double precision (64 bits)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent</th>
<th>significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

Largest = $1.111\ldots \times 2^{+1023}$

Smallest = $1.000\ldots \times 2^{-1024}$
**Overflow and underflow in FP**

An **overflow** occurs when the number is too large to fit in the frame. An **underflow** occurs when the number is too small to fit in the given frame.

**How do we represent zero?**

IEEE standards committee solved this by making **zero** a special case: if every bit is zero (the sign bit being irrelevant), then the number is considered zero.

**Then how do we represent 1.0?**
Then how do we represent 1.0?

It should have been $1.0 \times 2^0$ (same as 0)! The way out of this is that the interpretation of the exponent bits is not straightforward. The exponent of a single-precision float is "shift-127" encoded (biased representation), meaning that the actual exponent is (xxxxxxx minus 127). So thankfully, we can get an exponent of zero by storing 127.

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111 (i.e. 255)</td>
<td>255-127 = 128</td>
</tr>
<tr>
<td>0111111 (i.e. 127)</td>
<td>127-127 = 0</td>
</tr>
<tr>
<td>00000001 (i.e. 1)</td>
<td>1-127 = -126</td>
</tr>
</tbody>
</table>
**More on Biased Representation**

The consequence of shift-127

Exponent = 00000000 (reserved for 0) can no more be used to represent the smallest number. We forego something at the lower end of the spectrum of representable exponents, (which could be $2^{-127}$). That said, it seems wise, to give up the smallest exponent instead of giving up the ability to represent 1 or zero!
More special cases

Zero is not the only "special case" float. There are also representations for positive and negative infinity, and for a not-a-number (NaN) value, for results that do not make sense (for example, non-real numbers, or the result of an operation like infinity times zero). How do these work? A number is infinite if every bit of the exponent is 1 (yes, we lose another one), and is NaN if every bit of the exponent is 1 plus any mantissa bits are 1. The sign bit still distinguishes +/-inf and +/-NaN. Here are a few sample floating point representations:

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>0</td>
<td>Nonzero</td>
<td>Denormalized number*</td>
</tr>
<tr>
<td>1-254</td>
<td>Anything</td>
<td>+/- FP number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>Nonzero</td>
<td>NaN like 0/0 or ox inf</td>
</tr>
</tbody>
</table>

* Any non-zero number that is smaller than the smallest normal number is a denormalized number. The production of a denormal is sometimes called gradual underflow because it allows a calculation to lose precision slowly when the result is small.
Floating point operations in MIPS

32 separate single precision FP registers in MIPS

\[ f_0, f_1, f_2, \ldots f_{31}, \]

Can also be used as 16 double precision registers

\[ f_0, f_2, f_4, f_{30} \] (f0 means f0,f1 f2 means f2,f3)

These reside in a coprocessor C1 in the same package

Operations supported

add.\text{s} $f_2, f_4, f_6$ # $f_2 = f_4 + f_6$ (single precision)

add.\text{d} $f_2, f_4, f_6$ # $f_2 = f_4 + f_6$ (double precision)

(Also subtract, multiply, divide format are similar)

\text{lwc1} $f_1, 100(s_2)$ # $f_1 = M [s_2 + 100]$ (32-bit load)

\text{mtc1} $t_0, f_0$ # $f_0 = t_0$ (move to coprocessor 1)

\text{mfc1} $t_1, f_1$ # $t_1 = f_1$ (move from coprocessor 1)
Sample program

Evaluation of a Polynomial $a.x^2 + b.x + c$

```
# $f0 --- x
# $f2 --- sum of terms

# Evaluate the quadratic
l.s    $f2,a              # sum = a
mul.s  $f2,$f2,$f0         # sum = ax

l.s    $f4,b              # get b
add.s  $f2,$f2,$f4         # sum = ax + b
mul.s  $f2,$f2,$f0         # sum = (ax+b)x = ax^2 + bx

l.s    $f4,c              # get c
add.s  $f2,$f2,$f4         # sum = ax^2 + bx + c

.data
a:     .float  1.0
b:     .float  1.0
c:     .float  1.0
```

Pseudo-instruction
Floating Point Addition

Example using decimal

\[ A = 9.999 \times 10^1, B = 1.610 \times 10^{-1}, A+B =? \]

**Step 1.** Align the smaller exponent with the larger one.

\[ B = 0.0161 \times 10^1 = 0.016 \times 10^1 \text{ (round off)} \]

**Step 2.** Add significands

\[ 9.999 + 0.016 = 10.015, \text{ so } A+B = 10.015 \times 10^1 \]

**Step 3.** Normalize

\[ A+B = 1.0015 \times 10^2 \]

**Step 4.** Round off

\[ A+B = 1.002 \times 10^2 \]

Now, try to add 0.5 and -0.4375 in binary.
Floating Point Multiplication

Example using decimal

\[ A = 1.110 \times 10^{10}, \ B = 9.200 \times 10^{-5} \quad A \times B = ? \]

**Step 1.** Exponent of \( A \times B = 10 + (-5) = 5 \)

**Step 2.** Multiply significands

\[ 1.110 \times 9.200 = 10.212000 \]

**Step 3.** Normalize the product

\[ 10.212 \times 10^5 = 1.0212 \times 10^6 \]

**Step 4.** Round off

\[ A \times B = 1.021 \times 10^6 \]

**Step 5.** Decide the sign of \( A \times B \) (+ x + = +)

So, \( A \times B = + 1.021 \times 10^6 \)