Double-precision float numbers

<table>
<thead>
<tr>
<th>W1: Most Significant Word</th>
<th>W0: Least Significant Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29</td>
<td>22 21 20 19 18 ... 0</td>
</tr>
<tr>
<td>S</td>
<td>Ed=E+1023</td>
</tr>
<tr>
<td>Biased-Exponent (11 bits)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fd=F-1</td>
<td></td>
</tr>
<tr>
<td>Binary-Significand (52 bits)</td>
<td></td>
</tr>
</tbody>
</table>

where, $S$ is 1-bit sign, and it is set if the number is negative,

$Ed = E+1023$ is the biased-exponent to represent the exponents from $-1022$ to $1024$, ($E=-1023$ or $Ed=0$ is reserved for number-zero).

$Fd = F-1$ is the binary significand of the number, after the leading one bit in the normalized fraction of the number is removed.

The real number represented by a double-FP is

$$R = (-1)^S \times (Fd+1) \times 2^{Ed-1023}.$$

IEEE-754 double-precision-FP-fields | value of the number
---|---
$S$ (1-bit), $Ed$ (11-bit), $Fd$ (52-bit) | $R=(-1)^S \times (Fd+1) \times 2^{Ed-1023}$

Example: Given $-(0.75)_{10}$, show its MIPS representations in single and double precision

$-(0.75)_{10} = -(0.11)_2$

$= -(0.11)_2 \times 2^0$

$= -(1.1)_2 \times 2^{1}$ in normalized form

remember that the general representation for a fl.pt.no: $(-1)^S \times (1+F) \times 2^{(E-bias)}$

$\Rightarrow$ for single precision rep. $S=-1$, $F=.100...0$, $E-127=-1$, so $E=126$

31 30 29 22 21 20 19 18 1 0

S Exponent (E) Significand (F)

For the double precision rep., we have: $E-1023=-1$, so $E=1022$

31 30 29 22 21 20 19 18 1 0

S Exponent (E) [11 bits] Significand (F) [52 bits]

0 0 ...

Significand (continued)

Example: What decimal number is represented by the following MIPS fl.pt. rep.?

31 30 29 22 21 20 19 18 1 0

S Exponent (E) Significand (F)

a) $S=1 \Rightarrow$ no. is (-)
b) $F=1\times2^{-2} \equiv (0.25)_{10}$
c) $E=128+1=129$

general equation: no. = $(-1)^S \times (1+F) \times 2^{(E-bias)}$

$\Rightarrow$ no. = $-1 \times (1+0.25) \times 2^{(129-127)}$

$= -(1.25) \times 2^2$

$= -1.25 \times 4 = -5.0$
Special Cases
- The value zero
  - Defined to be the case where the significand and exponent are all 0’s
- In terms of floating point, overflow means the exponent is too large to fit
  Represent as the value infinity where exponent has all 1’s (11111111 = 255) and the significand is 0

\[
\begin{array}{c|c}
S & 1\ldots10\ldots0 \\
\end{array}
\]
- A new condition called underflow also exists where the exponent is too small to be represented
  - Represent as the value Not a Number (NaN) where exponent has all 1’s (11111111 = 255) and the significand is not 0

\[
\begin{array}{c|c}
S & 1\ldots1 Non\ Zero \\
\end{array}
\]

Floating-Point Addition:
By hand:  $9.999 \times 10^1 + 1.61 \times 10^{-1}$
Assume only 4 digits for the significand (F) and 2 digits for the exponent (E).
1. For proper addition, align the decimal point of the smaller number (i.e. shift the significand of the smaller number to the right as required):
   $1.61 \times 10^{-1} = 0.0161 \times 10^1$, but only 4 digits can be represented $\Rightarrow 0.016 \times 10^1$
2. Add the significands:
   \[
   \begin{array}{c}
   9.999 \\
   +0.016 \\
   \hline
   10.015 \\
   \end{array}
   \]
   So, the sum is $10.015 \times 10^1$
3. Normalize the sum:
   $10.015 \times 10^1 = 1.0015 \times 10^2$ (shift to the right)
   - Note that for a sum with leading 0’s, we have to shift it to the left.
   - Also, after every shift, make sure that there is no overflow or underflow
4. We have space for only 4 significand digits, so round the sum:
   \[
   1.0015 \times 10^2 \rightarrow 1.002 \times 10^2
   \]
   We may also choose to truncate the sum: $1.0015 \times 10^2 \rightarrow 1.001 \times 10^2$

IEEE-754 applies nearest-even rounding: A "one" is truncated if it is next to a zero, rounded-up if it is next to a one.
However, a zero is truncated in all cases.

Let's demonstrate four rounding modes on N= 1.0101 0011
- round up
  \[
  \begin{array}{c}
  N_1= 1.0101 010 ; N_2= 1.0101 01 ; N_3= 1.0101 1 ; N_4= 1.0110 ; \\
  \end{array}
  \]
- nearest-even
  \[
  \begin{array}{c}
  N_1= 1.0101 010 ; N_2= 1.0101 01 ; N_3= 1.0101 0 ; N_4= 1.0101 ; \\
  \end{array}
  \]
- truncate
  \[
  \begin{array}{c}
  N_1= 1.0101 001 ; N_2= 1.0101 00 ; N_3= 1.0101 0 ; N_4= 1.0101 ; \\
  \end{array}
  \]
Initialization: Load floating point numbers A and B.

1. Align the exponent of the smaller number to the exponent of the larger number.

2. Add the significands of the numbers

3. Normalize the sum, either shifting right and incrementing the exponent, or shifting left and decrementing the exponent.

   - Overflow or underflow: yes - exception
   - Normalized: no - yes, done

First the exponent of one operand is subtracted from the other using the Es-ALU to determine which is larger. This difference directs the control unit, to select the larger exponent, the significand of the smaller number, and significand of the larger number. The smaller significand is shifted right and then significands are added together using Fs-ALU. The normalization step then shifts the sum left or right and increases and decrements the exponent. Rounding then creates final result.
Example: Add 0.5 and -0.4375

\[(0.5)_{10} = 1/2 = (0.1)_2 \times 2^0 = 1.0 \times 2^{-1}\]
\[-(0.4375)_{10} = -7/16 = -0.0111_2 \times 2^3 = -1.11 \times 2^{-2}\]

**Step 1:** (Align the exponents): shift the significand of the smaller number \(-1.11 \times 2^{-2}\) right until its exponent matches the other number. ⇒ \(-1.11 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}\)

**Step 2:** Add the significands:

\[1.000 + (-0.111) = 1.000 + 1.001 = 0.001 \Rightarrow \text{the result is } 0.001 \times 2^{-1}\]

**Step 3:** Normalize the sum:

\[0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}\] (shift to the left)

Since \(-126 \leq -4 \leq 127\), there is no underflow or overflow

**Step 4:** Round the sum: it is not needed here

∴ the sum is \((1.000)_2 \times 2^{-4}\)

\[= (0.00010000)_2 = (0.0001)_2\]
\[= (1/2^4)_{10} = (1/16)_{10} = (0.0625)_{10}\]

Floating point multiplication algorithm:

1. Add the biased Exponents, subtract the bias from the sum to find the new biased exponent
2. Multiply the significands
3. Normalize the result: If it has leading 0’s, shift it to the left, and decrement E properly. If it has leading digits ≠0, shift it to the right, and increment E properly.
4. Round the significand if necessary
5. Set the sign bit of the product, by Exclusive ORing the sign bits of the operands

Stop
Example: Multiply $(0.5)_{10}$ by $(-0.4375)_{10}$ using the binary fl.pt. multiplication algorithm:

0. $(0.5)_{10} = (1.000)_{2} \times 2^{-1}$
   $- (0.4375)_{10} = -(1.110)_{2} \times 2^{-2}$

1. $(-1+127) + (-2+127) - 127 = 124$ (for $-3$)

2. 
   \[
   \begin{array}{c|cccc}
   & 1 & 0 & 0 & 0 \\
   \times & 1 & 1 & 1 & 0 \\
   \hline
   & 1 & 0 & 0 & 0 \\
   & 1 & 0 & 0 & 0 \\
   \hline
   & 1 & 1 & 1 & 0 & 0 & 0
   \end{array}
   \]
   \[
   \begin{array}{c|cccc}
   \end{array}
   \]
   ∴ The result is $(1.110)_{2} \times 2^{-3}$

3. It is already in normalized form. There is no overflow or under flow ($-126 \leq -3 \leq 127$)

4. No need for rounding

5. Signs of operands were different ⇒ sign of result is negative
   ∴ the product is $-(1.110)_{2} \times 2^{-3}$

Check the result by converting it to decimal:

$-(1.110)_{2} \times 2^{-3} = -0.001110 = -0.00111 = -7/2^{5} = -7/32 = -0.21875$

**MIPS FPU Instructions**

MIPS does not use general-purpose registers for FP arithmetic. It has 32 32-bit registers. $f0, f1, ..., f31$ for single and double-precision floating point numbers. The registers with odd numbers cannot be used in FP arithmetic instructions. In the instructions, the contents of the register-pairs are denoted by $f0, f2, ..., f30$ (only 16 registers for both single and double precision numbers). MIPS use special load and store instructions for memory-FP-register data transfer. There is no direct data transfer between $r0, ..., r31$ and $f0, ..., f31$. 
### MIPS FPU Instructions

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Instructions</strong></td>
<td><strong>Arithmetic Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><code>add.s</code> dest, src1, srcr2</td>
<td><code>add.d</code> dest, src1, srcr2</td>
<td>single/double FP addition</td>
</tr>
<tr>
<td><code>add.s</code> $f0,$f2,$f4</td>
<td><code>add.d</code> $f0,$f2,$f4</td>
<td>$f0 ← f2 + f4$</td>
</tr>
<tr>
<td><code>sub.s</code> dest, src1, src2</td>
<td><code>sub.d</code> dest, src1, src2</td>
<td>single/double FP subtraction</td>
</tr>
<tr>
<td><code>sub.s</code> $f0,$f2,$f4</td>
<td><code>sub.d</code> $f0,$f2,$f4</td>
<td>$f0 ← f2 + f4$</td>
</tr>
<tr>
<td><code>mul.s</code> dest, src1, src2</td>
<td><code>mul.d</code> dest, src1, src2</td>
<td>single/double FP multiplication</td>
</tr>
<tr>
<td><code>mul.s</code> $f0,$f2,$f4</td>
<td><code>mul.d</code> $f0,$f2,$f4</td>
<td>$f0 ← f2 + f4$</td>
</tr>
<tr>
<td><code>div.s</code> dest, src1, src2</td>
<td><code>div.d</code> dest, src1, src2</td>
<td>single/double FP division</td>
</tr>
<tr>
<td><code>div.s</code> $f0,$f2,$f4</td>
<td><code>div.d</code> $f0,$f2,$f4</td>
<td>$f0 ← f2 + f4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single precision transfer instructions</th>
<th>Double precision transfer instructions</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>l.s</code> dest, offset(base)</td>
<td><code>l.d</code> dest, offset(base)</td>
<td>single/double FP load</td>
</tr>
<tr>
<td><code>l.s</code> $f2,2000($4)</td>
<td><code>l.d</code> $f2,2000($4)</td>
<td>$f2 ← Memory(2000 + $4)$</td>
</tr>
<tr>
<td><code>s.s</code> src, offset(base)</td>
<td><code>s.d</code> src, offset(base)</td>
<td>single/double FP store</td>
</tr>
<tr>
<td><code>s.s</code> $f2,2000($4)</td>
<td><code>s.d</code> $f2,2000($4)</td>
<td>Memory(2000 + $4) ← $f2</td>
</tr>
</tbody>
</table>

**compare and branch**

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c.rel.s</code> src1, src2</td>
<td><code>c.rel.d</code> src1, src2</td>
</tr>
<tr>
<td><code>rel</code> may be</td>
<td><code>rel</code> may be</td>
</tr>
<tr>
<td><code>eq</code>, <code>neq</code>, <code>lt</code>, <code>le</code>, <code>gt</code>, <code>ge</code></td>
<td><code>eq</code>, <code>neq</code>, <code>lt</code>, <code>le</code>, <code>gt</code>, <code>ge</code></td>
</tr>
<tr>
<td>single-FP compare</td>
<td>double-FP compare</td>
</tr>
<tr>
<td>result sets cond-flag</td>
<td>result sets cond-flag</td>
</tr>
</tbody>
</table>

**Example**

For the C-language statements

```c
float a, b, c;
...
if(a<c) { a = a + b;}
```

compiler allocates three words in memory, at the locations `mema, memb, memc`. The assembler code corresponding to the addition statement is:

```assembly
l.s $f0,mema($0)    # $f0 = a
l.s $f2,memc($0)    # $f2 = c
c.lt.s $f0,$f2
bclf endif
l.s $f4,memb($0)    # $f4 = b
add.s $f0,$f0,$f4
s.s $f0,mema($0)
endif: ...
```

Note that the `FP-branch-on-condition` instruction shall be used after a `FP-compare`, which sets or resets the condition flag accordingly.