Representing Real Numbers

- **Floating Point** scheme.
- No universal convention (unlike 2’s complement for negative integers) yet.

**Scientific Notation:**
A real number is represented as

\[ (-1)^s \ m \times r^e \]

- **s:** Sign (0 or 1)
- **m:** Mantissa (Significand)
- **r:** Radix (Base)
- **e:** Exponent

**Example:** \[ 64.27 \times 10^7 \]

**Difficulty:** Many ways to represent the same number.

**IEEE FPS (Single precision):**

- **Word size = 32 bits** (corresponds to data type `float`).
- 1 bit used for sign, 8 bits for representing exponent and 23 bits for mantissa.
- **Positive range:** \(2 \times 10^{-38}\) to \(2 \times 10^{38}\) (approx).

\[
\begin{array}{c|c|c}
\text{Exponent} & \text{Sign bit} & \text{Mantissa} \\
\hline
E & s & M \\
23 & 1 & 23 \\
\end{array}
\]

**Sign bit:** This is the sign of the mantissa (sign = 0 for non-negative and 1 for negative).

**Exponent representation:** Uses 8-bit Biased-127 form; that is, if the actual exponent value is \(e\), it is stored as the 8-bit unsigned representation of the integer \(e + 127\).

**Example 1:** Suppose the actual exponent is \(-3\). It will be stored as the 8-bit representation of \(127 + (-3) = 124\), which is \(01111100\).
Example 2: Suppose the value stored in the exponent field is 1011 0101 binary. The stored value in decimal = 181. So, the actual exponent is $181 - 127 = 54$ decimal.

Mantissa representation: IEEE FPS for single precision allows 23 bits after the binary point.

The mantissa has the form:

$$1 . f_2 f_1 f_0 \ldots f_1 f_0$$

where each $f_i$ is 0 or 1.

Important note: Trailing zeros are added to the mantissa (if necessary) so that there are exactly 23 bits to the right of the binary point.

Example: The mantissa 1.101001 is expanded to 23 bits as follows.

$$1.101001000 \ldots 0 \text{ (17 trailing zeros)}$$

Step 4: Actual exponent = +3. So, in 8-bit biased-127 form, the exponent = 8-bit binary form of $127 + 3 = 130$ which is 1000 0010.

Step 5: So, IEEE FPS representation is:

$$\begin{array}{ll} 
  \hline 
  & 1 \times 2^{130} \\
  \hline 
\end{array}$$

Problem 2: Find the IEEE FPS single precision representation for the decimal number $-17.15$.

Solution:

Step 1: The binary representation for $-17.15 = -10001.001001$. (The binary representation is non-terminating.) The normalized binary representation is $-1.0001001001 \times 2^4$.

Step 2: The sign bit = 1.

Step 3: The significand is

$$0001 00 1001 1001 1001 1001 1$$

(Repeating part truncated when we have 23 bits total.)

The hidden bit: Since there is always a 1 to the left of the binary point, that bit is not stored. Only the 23 bits to the right of the binary point are stored. (These 23 bits form the significand.)

Let $F$ be given by

$$F = \sum_{i=0}^{22} f_i \times 2^i$$

The value of the mantissa $m$ is given by

$$m = 1 + F/2^{23}.$$ 

Problem 1: Find the IEEE FPS single precision representation for the decimal number 13.125.

Solution:

Step 1: The binary representation for 13.125 = 1101.001. The normalized binary representation is $1.101001 \times 2^3$.

Step 2: The sign bit = 0.

Step 3: The significand = 101001000 \ldots 0 (with 17 trailing zeros).

Step 4: Actual exponent = +4. So, in 8-bit biased-127 form, the exponent = 8-bit binary form of $127 + 4 = 131$ which is 1000 0011.

Step 5: So, IEEE FPS representation is:

$$\begin{array}{ll} 
  \hline 
  & 1 \times 2^{131} \\
  \hline 
\end{array}$$

Other considerations in IEEE FPS:

(a) Representing zero: Stored exponent = 0 and Significand = 0. (Convention does not specify sign bit. So, there are two representations for zero.)

One consequence is that the least possible exponent (in single precision) is $-126$ (represented as 0000 0001). The exponent $-127$ is not allowed.

(b) Representing $+\infty$ (floating overflow):

Sign = 0. Stored exponent = 255 (decimal) and Significand = 0.
(c) Representing $-\infty$ (floating underflow):
Sign = 1, Stored exponent = 255 (decimal) and Significand = 0.

(d) Result of undefined operation: (E.g. 0/0)
Stored exponent = 255 (decimal) and Significand $\neq 0$. (The value is denoted by NaN.)

IEEE FPS (Double precision):
- Word size = 64 bits (corresponds to data type double).
- 1 bit used for sign, 11 bits for representing exponent and 52 bits for significand.
- Exponent stored in 11-bit biased-1023 form.
- Positive range: $2 \times 10^{-308}$ to $2 \times 10^{308}$ (approx).

Why biased representation for exponent:
- Suitable for processing floating point numbers using integer operations. (E.g. comparison of two floating point numbers.)

- If 2's complement is used, negative exponents will have sign bit 1. The exponent will look like a large number.
- Desirable notation should use 0000 0000 for the smallest (i.e., most negative) exponent and 1111 1111 for the largest (i.e., most positive) exponent. (This is exactly what biased representation provides.)

MIPS hardware support for FPS:
- MIPS supports both single precision and double precision floating point numbers using FPS.
- Separate set of 16 registers for storing floating point operands (part of Coprocessor 1).
- Floating point registers denoted by $f0$, $f2$, ..., $f30$. (No odd numbered registers.)

- When single precision is used, $f0$ is a register of size 32 bits. When double precision is used, $f0$ is a register of size 64 bits.
- In opcodes suffix ".s" indicates single precision and ".d" indicates double precision.

Some floating point instructions:
(a) Arithmetic operations:
- add.s, sub.s, mul.s, div.s.
- add.d, sub.d, mul.d, div.d.

Examples:
add.s $f2$, $f8$, $f10$
sub.d $f4$, $f10$, $f16$

(b) Load/Store operations:
- l.s, s.s.
- l.d, s.d.
- li.s, li.d.

Examples:
l.s $f16$, 20($9$
s.d $f4$, 80($13$
li.s $f6$, -17.25
li.d $f20$, 0.002

(c) Move operations:
- mov.s, mov.d.

Examples:
mov.s $f16$, $f8$
mov.d $f18$, $f10$
(d) Comparison operations:
- c lt.s, c lt.d.
- Instead of “lt”, can also have eq, ne, le, gt and ge.
- Result of comparison sets flag (called FP) in Coprocessor 1.

Examples:

- c lt.s  $f16, $f8
- c ge.d  $f18, $f10

(e) Branch operations:
- bc1t, bc1f.
- Branch happens if the FP flag is set to true (1) or false (0).

Examples:

- bc1t  check_val
- bc1f  get_result

(i) int to float or double:
- cvt.s.w, cvt.d.w.

Important note: Both registers specified in any cvt instruction must be Coprocessor 1 registers.
Example: To convert the integer (in 2's complement form) in $f10 into float and store the result in $f8, the instruction is:

\[
\text{cvt.s.w} \quad $f8, $f10
\]

Use: A C++ compiler would use the above instruction in compiling statements such as the following:

\[
\text{int} \ i = 17; \ \text{float} \ x; \\
\text{x} = \text{float}(i);
\]

If $i$ is in $9$ and $x$ is in $f10$, the MAL code for the assignment statement is:

\[
\begin{align*}
\text{mtc1} & \quad $9, $f6 \\
\text{cvt.s.w} & \quad $f10, $f6
\end{align*}
\]

(f) Moving from/to Coprocessor 1 Registers:
- mfc1, mtc1.

Examples:

\[
\begin{align*}
\text{mfc1} & \quad $5, $f10 \ \#\text{Destination:} \ $5 \\
\text{mtc1} & \quad $9, $f12 \ \#\text{Destination:} \ $f12
\end{align*}
\]

Note: In both the instructions mfc1 and mtc1, the first register is the usual CPU register and the second is the Coprocessor 1 register.

(g) Conversions among int, float and double:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Single precision (float)</td>
</tr>
<tr>
<td>d</td>
<td>Double precision (double)</td>
</tr>
<tr>
<td>w</td>
<td>Word (int)</td>
</tr>
</tbody>
</table>

(ii) float or double to int:
- cvt.w.s, cvt.w.d.
- Conversion involves truncation.
- Overflow/underflow can occur.
A C++ compiler would use these instructions to translate a typecast from float or double to int.

Examples:

\[
\begin{align*}
\text{cvt.w.s} & \quad $f10, $f4 \\
\text{cvt.w.d} & \quad $f14, $f6
\end{align*}
\]

(iii) float to double or vice versa:
- cvt.d.s, cvt.s.d.
- cvt.s.d may cause overflow/underflow.

Examples:

\[
\begin{align*}
\text{cvt.d.s} & \quad $f8, $f2 \\
\text{cvt.s.d} & \quad $f14, $f6
\end{align*}
\]
I/O for floating point numbers:

- Can use syscall.
- Integer command for the operation specified in $v0 (as in the case of integer I/O).
- For an input operation, the value is returned in $f0.
- For an output operation, the value to be output must be in $f12.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_float</td>
<td>2</td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
</tr>
</tbody>
</table>

Examples:

1) Reading a value from keyboard:

```assembly
li $v0, 6      #read_float command
syscall        #Value in $f0.
```

2) Printing a value:

```assembly
.data
v1: .float -17.15
.text
la $9, v1
l.s $f12, 0($9)
li $v0, 2      #print_float command
syscall        #Value in $f12.
```

MAL program example: Handout.