EE 109 Unit 20

IEEE 754 Floating Point Representation

Floating Point Arithmetic
Floating Point

• Used to represent very small numbers (fractions) and very large numbers
  – Avogadro’s Number: $+6.0247 \times 10^{23}$
  – Planck’s Constant: $+6.6254 \times 10^{-27}$
  – Note: 32 or 64-bit integers can’t represent this range

• Floating Point representation is used in HLL’s like C by declaring variables as `float` or `double`
Fixed Point

- Unsigned and 2’s complement fall under a category of representations called “Fixed Point”
- The radix point is assumed to be in a fixed location for all numbers [Note: we could represent fractions by implicitly assuming the binary point is at the left...A variable just stores bits...you can assume the binary point is anywhere you like]
  - Integers: \(10011101\). (binary point to right of LSB)
    - For 32-bits, unsigned range is 0 to ~4 billion
  - Fractions: \(0.10011101\) (binary point to left of MSB)
    - Range [0 to 1]

- **Main point**: By fixing the radix point, we limit the range of numbers that can be represented
  - Floating point allows the radix point to be in a different location for each value
Floating Point Representation

• Similar to scientific notation used with decimal numbers
  – \( \pm D.DDD \times 10^{\pm \text{exp}} \)

• Floating Point representation uses the following form
  – \( \pm b.bbbb \times 2^{\pm \text{exp}} \)
  – 3 Fields: sign, exponent, fraction (also called mantissa or significand)

<table>
<thead>
<tr>
<th>S</th>
<th>Exp.</th>
<th>fraction</th>
</tr>
</thead>
</table>

Overall Sign of #
Normalized FP Numbers

• Decimal Example
  – +0.754*10^{15} is not correct scientific notation
  – Must have exactly one significant digit before decimal point:
    +7.54*10^{14}

• In binary the only significant digit is ‘1’

• Thus normalized FP format is:
  \[ \pm 1.bbbbb * 2^{\pm \text{exp}} \]

• FP numbers will always be normalized before being stored in memory or a reg.
  – The 1. is actually not stored but assumed since we always will store normalized numbers
  – If HW calculates a result of 0.001101*2^5 it must normalize to 1.101000*2^2 before storing
## IEEE Floating Point Formats

- **Single Precision** (32-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - 8 Exponent bits
    - Excess-127 representation
    - More on next slides
  - 23 fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 7 digits x $10^{±38}$

<table>
<thead>
<tr>
<th>S</th>
<th>Exp.</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Double Precision** (64-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - 11 Exponent bits
    - Excess-1023 representation
    - More on next slides
  - 52 fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 16 digits x $10^{±308}$

<table>
<thead>
<tr>
<th>S</th>
<th>Exp.</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exponent Representation

• Exponent needs its own sign (+/-)
• Rather than using 2’s comp. system we use Excess-N representation
  – Single-Precision uses Excess-127
  – Double-Precision uses Excess-1023
  – This representation allows FP numbers to be easily compared
• Let $E' = \text{stored exponent code}$ and $E = \text{true exponent value}$
• For single-precision: $E' = E + 127$
  – $2^1 \Rightarrow E = 1, E' = 128_{10} = 10000000_2$
• For double-precision: $E' = E + 1023$
  – $2^{-2} \Rightarrow E = -2, E' = 1021_{10} = 0111111101_2$

<table>
<thead>
<tr>
<th>2’s comp.</th>
<th>E' (stored Exp.)</th>
<th>Excess-127</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1111 1111</td>
<td>+128</td>
</tr>
<tr>
<td>-2</td>
<td>1111 1110</td>
<td>+127</td>
</tr>
<tr>
<td>-128</td>
<td>1000 0000</td>
<td>1</td>
</tr>
<tr>
<td>+127</td>
<td>0111 1111</td>
<td>0</td>
</tr>
<tr>
<td>+126</td>
<td>0111 1110</td>
<td>-1</td>
</tr>
<tr>
<td>+1</td>
<td>0000 0001</td>
<td>-126</td>
</tr>
<tr>
<td>0</td>
<td>0000 0000</td>
<td>-127</td>
</tr>
</tbody>
</table>

Comparison of 2’s comp. & Excess-N

Q: Why don’t we use Excess-N more to represent negative #’s
Exponent Representation

- FP formats reserved the exponent values of all 1’s and all 0’s for special purposes.
- Thus, for single-precision the range of exponents is -126 to +127.

<table>
<thead>
<tr>
<th>E'</th>
<th>E (E' - 127) and special values</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>Reserved</td>
</tr>
<tr>
<td>11111110</td>
<td>E'-127=+127</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>E'-127=+1</td>
</tr>
<tr>
<td>01111111</td>
<td>E'-127=0</td>
</tr>
<tr>
<td>01111110</td>
<td>E'-127=-1</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>E'-127=-126</td>
</tr>
<tr>
<td>00000000</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
## IEEE Exponent Special Values

<table>
<thead>
<tr>
<th>E’</th>
<th>Fraction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>All 0’s</td>
<td>All 0’s</td>
<td>0</td>
</tr>
<tr>
<td>All 0’s</td>
<td>Not all 0’s (any bit = ‘1’)</td>
<td>Denormalized (0.fraction x (2^{-126}))</td>
</tr>
<tr>
<td>All 1’s</td>
<td>All 0’s</td>
<td>Infinity</td>
</tr>
<tr>
<td>All 1’s</td>
<td>Not all 0’s (any bit = ‘1’)</td>
<td>NaN (Not A Number) - (0/0, 0^\infty, \sqrt{-x})</td>
</tr>
</tbody>
</table>
Single-Precision Examples

1. \[1 1000 0010 \quad 110 0110 0000 0000 0000 0000\]

\[
130 - 127 = 3
\]

\[-1.1100110 \times 2^3\]

\[= -1110.011 \times 2^0\]

\[= -14.375\]

2. \[+0.6875 = +0.1011\]

\[= +1.011 \times 2^{-1}\]

\[-1 + 127 = 126\]
Floating Point vs. Fixed Point

• Single Precision (32-bits) Equivalent Decimal Range:
  – 7 significant decimal digits * 10\(^{±38}\)
  – Compare that to 32-bit signed integer where we can represent ±2 billion. How does a 32-bit float allow us to represent such a greater range?
  – FP allows for range but sacrifices precision (can’t represent all numbers in its range)

• Double Precision (64-bits) Equivalent Decimal Range:
  • 16 significant decimal digits * 10\(^{±308}\)
IEEE Shortened Format

- 12-bit format defined just for this class (doesn’t really exist)
  - 1 Sign Bit
  - 5 Exponent bits (using Excess-15)
    - Same reserved codes
  - 6 Fraction (significand) bits
Examples

1

<table>
<thead>
<tr>
<th></th>
<th>10100</th>
<th>101101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20-15=5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.101101 * 2^5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>= -110110.1 * 2^0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>= -110110.1 = -54.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2

<table>
<thead>
<tr>
<th></th>
<th>+21.75 = +10101.11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= +1.010111 * 2^4</td>
</tr>
<tr>
<td></td>
<td>4+15=19</td>
</tr>
<tr>
<td></td>
<td>0 10011 010111</td>
</tr>
</tbody>
</table>

3

<table>
<thead>
<tr>
<th></th>
<th>01101</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13-15=-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.100000 * 2^-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>= -0.011 * 2^0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>= -0.011 = -0.375</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4

<table>
<thead>
<tr>
<th></th>
<th>+3.625 = +11.101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= +1.110100 * 2^1</td>
</tr>
<tr>
<td></td>
<td>1+15=16</td>
</tr>
<tr>
<td></td>
<td>0 10000 110100</td>
</tr>
</tbody>
</table>
Rounding Methods

- +213.125 = 1.1010101001*2^7 => Can’t keep all fraction bits
- 4 Methods of Rounding (you are only responsible for the first 2)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to Nearest</td>
<td>Normal rounding you learned in grade school. Round to the nearest representable number. If exactly halfway between, round to representable value w/ 0 in LSB.</td>
</tr>
<tr>
<td>Round towards 0 (Chopping)</td>
<td>Round the representable value closest to but not greater in magnitude than the precise value. Equivalent to just dropping the extra bits.</td>
</tr>
<tr>
<td>Round toward +∞ (Round Up)</td>
<td>Round to the closest representable value greater than the number</td>
</tr>
<tr>
<td>Round toward -∞ (Round Down)</td>
<td>Round to the closest representable value less than the number</td>
</tr>
</tbody>
</table>
Number Line View Of Rounding Methods

Green lines are numbers that fall between two representable values (dots) and thus need to be rounded.

Round to Nearest

Round to Zero

Round to +Infinity

Round to -Infinity
Rounding Implementation

- There may be a large number of bits after the fraction
- To implement any of the methods we can keep only a subset of the extra bits after the fraction [hardware is finite]
  - Guard bits: bits immediately after LSB of fraction (in this class we will usually keep only 1 guard bit)
  - Round bit: bit to the right of the guard bits
  - Sticky bit: Logical OR of all other bits after G & R bits

\[
\begin{align*}
1.010010 & \quad 10010 \quad \times \quad 2^4 \\
\downarrow \downarrow \downarrow \quad \text{Logical OR (output is ‘1’ if any input is ‘1’, ‘0’ otherwise)}
\end{align*}
\]

\[
\begin{align*}
1.010010 & \quad 101 \quad x \quad 2^4 \\
\text{GRS}
\end{align*}
\]

We can perform rounding to a 6-bit fraction using just these 3 bits.
Rounding to Nearest Method

• Same idea as rounding in decimal
  – .51 and up, round up,
  – .49 and down, round down,
  – .50 exactly we round up in decimal
    • In this method we treat it differently...If precise value is exactly half way between 2 representable values, round towards the number with 0 in the LSB
Round to Nearest Method

- Round to the closest representable value
  - If precise value is exactly half way between 2 representable value, round towards the number with 0 in the LSB

\[
\begin{align*}
1.11111011010 \times 2^4 \\
1.111110111 \times 2^4 \\
\text{GRS}
\end{align*}
\]

Precise value will be rounded to one of the representable value it lies between.

In this case, round up because precise value is closer to the next higher representable values
Rounding to Nearest Method

• 3 Cases in binary FP:
  – G = ‘1’ & (R,S ≠ 0,0) =>
    • round fraction up (add 1 to fraction)
    • may require a re-normalization
  – G = ‘1’ & (R,S = 0,0) =>
    • round to the closest fraction value with a ‘0’ in the LSB
    • may require a re-normalization
  – G = ‘0’ =>
    • leave fraction alone (add 0 to fraction)
Round to Nearest

1.001100 \times 2^4

G = '1' & R, S \neq 0, 0

Round up (fraction + 1)

G = '0'

Leave fraction

1.1111110 x 2^4

G = '1' & R, S \neq 0, 0

Round up (fraction + 1)

1.0011010 x 2^4

G = '0'

Leave fraction

\[
\begin{align*}
1.111111 & \times 2^4 \\
+ 0.000001 & \times 2^4 \\
\hline
10.000000 & \times 2^4 \\
1.000000 & \times 2^5 \\
\hline
0 & 10100 \quad 000000
\end{align*}
\]

Requires renormalization
Round to Nearest

- In all these cases, the numbers are halfway between the 2 possible round values
- Thus, we round to the value w/ 0 in the LSB

<table>
<thead>
<tr>
<th>GRS</th>
<th>(1.001100) (\times) (2^4)</th>
<th>GRS</th>
<th>(1.111111) (\times) (2^4)</th>
<th>GRS</th>
<th>(1.001101) (\times) (2^4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G = ‘1’ and R,S = ‘0’</td>
<td>Rounding options are: (1.001100) or (1.001101)</td>
<td>G = ‘1’ and R,S = ‘0’</td>
<td>Rounding options are: (1.111111) or (10.000000)</td>
<td>G = ‘1’ and R,S = ‘0’</td>
<td>Rounding options are: (1.001101) or (1.001110)</td>
</tr>
</tbody>
</table>

In this case, round down

\[
\begin{array}{ccc}
0 & 10011 & 001100 \\
\end{array}
\]

\[
\begin{array}{ccc}
1.111111 & \times & 2^4 \\
0.000001 & \times & 2^4 \\
\hline
10.000000 & \times & 2^4 \\
1.000000 & \times & 2^5 \\
\hline
0 & 10100 & 000000 \\
\end{array}
\]

Requires renormalization
Round to 0 (Chopping)

- Simply drop the G,R,S bits and take fraction as is

\[
\begin{align*}
1.001100001 \times 2^4 & \quad \text{drop G,R,S bits} \quad 0 \quad 10011 \quad 001100 \\
1.001101101 \times 2^4 & \quad \text{drop G,R,S bits} \quad 0 \quad 10011 \quad 001101 \\
1.001100111 \times 2^4 & \quad \text{drop G,R,S bits} \quad 0 \quad 10011 \quad 001100
\end{align*}
\]
Important Warning For Programmers

• FP addition/subtraction is NOT associative
  – Because of rounding / inability to precisely represent fractions, \((a+b)+c \neq a+(b+c)\)

\[(\text{small} + \text{LARGE}) - \text{LARGE} \neq \text{small} + (\text{LARGE} - \text{LARGE})\]

Why? Because of rounding and special values like \(\text{Inf}\).

\[(0.0001 + 98475) - 98474 \neq 0.0001 + (98475-98474)\]

\[98475-98474 \neq 0.0001 + 1\]

\[1 \neq 1.0001\]

Another Example:

\[1 + 1.11...1*2^{127} - 1.11...1*2^{127}\]
USING INTEGERS TO EMULATE DECIMAL/FRACTION ARITHMETIC
Option 1 for Performing FP

- Problem: Suppose you need to add, subtract and compare numbers representing FP numbers
  - Ex. amounts of money (dollars and cents.)
- Option 1: Use floating point variables
  - Pro: Easy to implement and easy to work with
  - Con: Some processors like Arduino don't have HW support of FP and so they would need to include a lot of code to emulate FP operations
  - Con: Numbers like $12.50 can be represented exactly but most numbers are approximate due to rounding of fractions that can be can't be represented in a finite number of bits.
    - Example 0.1 decimal can't be represented exactly in binary

```c
float x, y, z;
if (x > y)
    z = z + x;
```

Option 1: Just use 'float' or 'double' variables in your code
Option 2 for Performing FP

- Option 2: Split the amounts into two fixed-point variables, one for the dollars and one for the cents. $12.53 -> 12 and 53
  - Pro: Everything done by fixed-point operations, no FP.
  - Cons: All operations require at least two fixed point operations (though this is probably still faster than the code the compiler would include in Option 1)

```c
/* add two numbers */
z_cents = x_cents + y_cents;
z_dollars = x_dollars + y_dollars;
if (z_cents > 100) {
    z_dollars++;
    z_cents -= 100;
}
```

Option 2: Use 'ints' to emulate FP
Option 3 for Performing FP

- Option 3: Use a single fixed-point variable by multiplying the amounts by 100 (e.g. $12.53 -> 1253)
  - Pro: All adds, subtracts, and compares are done as a single fixed-point operation
  - Con: Must convert to this form on input and back to dollars and cents on output.

```cpp
int x_amount = x_dollars*100 + x_cents;
int y_amount = y_dollars*100 + y_cents;
int z_amount;
if (x_amount > y_amount) {
    z_amount += z_amount + x_amount;
}
int z_dollars = z_amount / 100;
int z_cents = z_amount % 100;
```

Option 3: Use single 'int' values that contain the combined values of integer and decimal
Emulating Floating Point

- Let's examine option 3 a bit more
- Suppose we have separate variables storing the integer and fraction part of a number separately
  - If we just added integer portions we'd get 18
  - If we account for the fractions into the inputs of our operation we would get 19.25 (which could then be truncated to 19)
  - We would prefer this more precise answer that includes the fractional parts in our inputs

Desired Decimal Operation
(separate integer / fraction):

<table>
<thead>
<tr>
<th>Integer</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.</td>
<td>75</td>
</tr>
<tr>
<td>6.</td>
<td>5</td>
</tr>
</tbody>
</table>

**Integer**  **Fraction**
Example and Procedure

- Imagine we took our numbers and multiplied them each by 100, performed the operation, then divided by 100
  - \(100 \times (12.75 + 6.5) = 1275 + 650 = 1925 \Rightarrow \frac{1925}{100} = 19.25 \Rightarrow 19\)

- Procedure
  - Assemble int + frac pieces into 1 variable
    - Shift integer to the left, then add/OR in fraction bits
  - Perform desired arithmetic operation
  - Disassemble int + frac by shifting back

\[
\begin{array}{|c|c|}
\hline
\text{Desired Decimal Operation} & \text{Shift Integers & Add in fractions:} & \text{Disassemble integer and fraction results (if only integer is desired, simply shift right to drop fraction)} \\
\hline
12.75 & 1275.0 & 19.25 \\
6.5 & 650.0 & 19.25 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Integer} & \text{Fraction} \\
\hline
12. & 75 \\
6. & 5 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Integer + Fraction} & \text{Integer + Fraction} \\
\hline
19. & 25 \\
19. & \text{Or just integer} \\
\hline
\end{array}
\]
Another Example

- Suppose we want to convert from "dozens" to number of individual items (i.e. 1.25 dozen = 15 items)
  - Simple formula: 12*dozens = individual items
  - Suppose we only support fractions: $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ represented as follows:

<table>
<thead>
<tr>
<th>Decimal View</th>
<th>Binary View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Fraction</td>
</tr>
<tr>
<td>3.</td>
<td>25</td>
</tr>
<tr>
<td>3.</td>
<td>50</td>
</tr>
<tr>
<td>3.</td>
<td>75</td>
</tr>
</tbody>
</table>
Another Example

• Procedure
  – Assemble int + frac pieces into 1 variable
    • Shift integer to the left, then add/OR in fraction bits
  – Perform desired arithmetic operation
  – Disassemble int + frac by shifting back

```
Decimal View

<table>
<thead>
<tr>
<th>i</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i = i &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>13.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i * 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>156.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i = i &gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>39.</td>
</tr>
</tbody>
</table>

Binary View

<table>
<thead>
<tr>
<th>i</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.</td>
<td>01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i = i &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100.</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>1101.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i * 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011100.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i = i &gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100111.</td>
</tr>
</tbody>
</table>
```