Floating Point

- Used to represent very _________ numbers (fractions) and very _________ 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 _____________________
- The radix point is _____________ 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: `.10011101` (binary point to left of MSB)
    - Range [0 to 1]
- Main point: By fixing the radix point, we __________ 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 ____________________ used with decimal numbers
- Floating Point representation uses the following form
  - $\pm b.bbbb \times 2^{\pm\text{exp}}$
  - 3 Fields: ________, ____________, __________ (also called __________ or significand)
Normalized FP Numbers

- **Decimal Example**
  - +0.754*10^{15} is ________ correct scientific notation
  - Must have exactly one ________________ before decimal point: ________________
- **In binary the only significant digit is ____________**
- **Thus normalized FP format is:**
  - FP numbers will always be ______________ before being stored in memory or a reg.
    - The ______ 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)
    - ___ Sign bit (0=pos/1=neg)
    - ___ Exponent bits
      - _______ representation
      - More on next slides
    - ___ fraction (significand or mantissa) bits
    - Equiv. Decimal Range:
      - 7 digits \( \times 10^{+38} \)

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

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 \( E' = \) stored exponent code and \( E = \) true exponent value
- For single-precision: \( E' = E + 127 \)
  - \( 2^1 \Rightarrow E = 1, E' = 128_{10} = 10000000_{2} \)
- For double-precision: \( E' = E + 1023 \)
  - \( 2^2 \Rightarrow E = -2, E' = 1021_{10} = 0111111101_{2} \)

**Comparison of 2’s comp. & Excess-N**

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

<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>

**E’ (range of 8-bits shown) and special values**

<table>
<thead>
<tr>
<th>E’</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
</tr>
<tr>
<td>11111110</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>10000000</td>
</tr>
<tr>
<td>01111111</td>
</tr>
<tr>
<td>01111110</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>00000001</td>
</tr>
<tr>
<td>00000000</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single-Precision Examples

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

2. +0.6875 = +0.1011

Floating Point vs. Fixed Point

- Single Precision (32-bits) Equivalent Decimal Range:
  - 7 significant decimal digits * \(10^{\pm38}\)
  - 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 \[\underline{\text{___________}}\] but sacrifices \[\underline{\text{___________}}\] (can't represent \[\underline{\text{___________}}\] in its range)
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits * \(10^{\pm308}\)

IEEE Shortened Format

- 12-bit format defined just for this class (doesn’t really exist)
  - 1 Sign Bit
  - 5 Exponent bits (using \[\underline{\text{___________}}\])
  - Same reserved codes
  - 6 Fraction (significand) bits

\[\begin{array}{cccc}
S & E' & F \\
\hline
1 & 5\text{-bits} & 6\text{-bits} \\
\end{array}\]
### Examples

1. $1 \ 1010 \ 101101$
2. $+21.75 = +10101.11$
3. $1 \ 01101 \ 100000$
4. $+3.625 = +11.101$

### Rounding Methods

- $+213.125 = 1.101010101001^27$ => 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 ______</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 ___ (__________)</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]
  - _____ bits: bits immediately after LSB of fraction (in this class we will usually keep only _____________ bit)
  - _____ bit: bit to the right of the guard bits
  - _____ bit: _____________ of all other bits after G & R bits

\[
\begin{align*}
1.01001010010 & \times 2^4 \\
1.0100101 & \times 2^4 \\
\text{GRS}
\end{align*}
\]

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

3 Cases in binary FP:
- \( G = \ldots \) =>
  - round fraction up (add 1 to fraction)
  - may require a re-normalization
- \( G = \ldots \) =>
  - round to the closest fraction value with a ‘0’ in the LSB
  - may require a re-normalization
- \( G = \ldots \) =>
  - leave fraction alone (add 0 to fraction)
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

\[
\begin{align*}
&1.001100 \times 2^4 \\
&1.111111 \times 2^4 \\
&1.001101 \times 2^4
\end{align*}
\]

Round to 0 (Chopping)

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

\[
\begin{align*}
&1.001100 \times 2^4 \\
&1.001101 \times 2^4 \\
&1.001100 \times 2^4
\end{align*}
\]

Important Warning For Programmers

- FP addition/subtraction is NOT __________
  - Because of rounding / inability to precisely represent fractions, (a+b)+c ≠ a+(b+c)

(small + LARGE) – LARGE ≠ small + (LARGE – LARGE)

Why? Because of _____________ and special values like Inf.

(0.0001 + 98475) – 98474 ≠ 0.0001 + (98475-98474)

98475-98474 ≠ 0.0001 + 1

1 ≠ 1.0001

Another Example:

\[
1 + 1.11\ldots 1*2^{127} - 1.11\ldots 1*2^{127}
\]

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 _____________ variables
  - Pro: Easy to implement and easy to work with
  - Con: Some processors like Arduino don't have __________ of FP and so they would need to include a lot of code to _________ FP operations
  - Con: Numbers like $12.50 can be represented exactly but most numbers are __________ 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

```
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 _____________ 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)

```
/* 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 ________ 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.

```
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
Example and Procedure

• Imagine we took our numbers and multiplied them each by 100, performed the operation, then divided by 100
  – $100 \times (12.75+6.5) = 1275 + 650 = 1925 \Rightarrow 1925/100 = 19.25 \Rightarrow 19$

• Procedure
  – Assemble __________ pieces into ____ variable
    • _______ integer to the _____, then ________ in fraction bits
  – Perform desired __________________________
  – __________ int + frac by __________ back

Desired Decimal Operation (separate integer / fraction):

<table>
<thead>
<tr>
<th></th>
<th>Shift Integers &amp; Add in fractions:</th>
<th>Disassemble integer and fraction results (if only integer is desired, simply shift right to drop fraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Decimal Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(separate integer / fraction):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>12.5</td>
<td>Integer + Fraction</td>
</tr>
<tr>
<td>Fraction</td>
<td></td>
<td>Or just Integer</td>
</tr>
<tr>
<td>+</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Another Example

• Suppose we want to convert from "dozens" to number of individual items (i.e. 1.25 dozen = 15 items)
  – Simple formula: $12 \times \text{dozens} = \text{individual items}$
  – 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>3.0</td>
<td>.25</td>
</tr>
<tr>
<td>3.50</td>
<td>.50</td>
</tr>
<tr>
<td>3.75</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