COMP253 — Lecture 7: **Floating-Point Fractional Values**

In representing fractional values, fixed-point is very limited in its range because of its very nature: the magnitude and precision are fixed.

Specifically, the drawbacks of fixed-point are,

- does not allow for the possibility of borrowing bits from the magnitude or precision to accommodate very large, or very small, values.
- possibility of wasted bits from values that are either primarily integer (wasted trailing "zeroes"), or primarily fractional (wasted leading "zeroes")
- inflexibility in calculations leading to overflow (loss of magnitude) and underflow (loss of precision)

Although fixed-point is simpler for humans to understand and manipulate, these drawbacks have required CPUs to be designed around another technique for representing fractional values. This form is called floating-point, and applies the concept of *scientific notation* (or *exponential notation*) binary representation.

Review: Scientific-Notation in Decimal (SN)

In this notation, the attempt is to store as much of the *magnitude* and *precision* of the decimal value as possible, without having to worry about leading or trailing zeros (although in proper-science, trailing zeroes usually define the "precision" of the recorded number). Further, all values are represented in the same strict format regardless of how large or small the *true value* is.

The SN form involves, *"reducing, or increasing, the magnitude of a* true value *to a common fractional value and storing the value of the reduction, or increase, as the power to the base."*

The following examples indicate how a *true value* is represented in Scientific Notation, both in the *common* form and in the *normalised* form.

common Scientific Notation)		normalised Scientific Notation	
0.4	$= 4.0 * 10^{-1}$	0.4	$= 0.4 * 10^{-0}$
0.05	$= 5.0 * 10^{-2}$	0.05	$= 0.5 * 10^{-1}$
1.0	$= 1.0 * 10^{0}$	1.0	$= 0.1 * 10^{1}$
154.245	$= 1.54245 * 10^{2}$	154.245	$= 0.154245 * 10^{3}$
0.0050013	$= 5.0013 * 10^{-3}$	0.0050013	$= 0.50013 * 10^{-2}$

The only difference between the two forms is the position of the "point." Although the common form seems natural, since all numbers always have a single integer digit, the advantage of the normalised form is that the **0.** at the beginning is not actually required and written only clarity—with a little more thought, it is easy to see that in *normalised* form, the **0.** is *not actually required!*

> ***Note:*** *In mathematics, the "point" is referred to as the "radix point," indicating where the unit value of the radix (or base) shifts between* whole number unit values *(where the power is positive) and* fractional unit values *(where the power is negative).*

Whether common or normalised, a representation in Scientific Notation has the following aspects of the true value,

1. a mantissa (or coefficient); example: 0.05 => 5.0 (common); or 154.245 => 0.154245 (normalised)
2. a exponent (magnitude of true value); example: 0.05 => 10^{-2} (common); or 154.245 => 10^{3} (normalised)

Afterwards, what were previously written as small or very large numbers, are all represented in the same format.

Extra Note: Math Co-Processors

CPUs are not able to store values directly in binary floating-point, as they can with integers.

Although floating-point (fp) is an elegant solution to the problems of fixed-point, it requires complex processor calculations to store, retrieve, and process fp numbers. Such calculations are usually done with within special numeric processing circuits.

To store, manipulate, and transfer fp values, CPUs were provided with "floating-point algorithms" (or routines) in special chips called *math co-processors* (or *numeric co-processors*). In modern processors, these routines are included within the single CPU, but still considered "co-".

Common names: math co-processors (MathCoPro), numeric processing units (NPU), or floating-point units (FPU).

> ***Note:*** *For CPUs that do not have dedicated NPU circuits, an emulation program must be used, in which the CPU must perform the function of the NPU along with the CPU's own instructions. This emulation is clearly time consuming and slows the CPU dramatically.*

[Binary] Floating-Point

The idea of scientific notation is defined in binary as floating-point representation, since the "point" seems to "float" from its original position in the *true value*, to the left most part of the number.

A binary floating-point number is defined by three (3) aspects:

- a sign bit, to indicate the sign of the mantissa (i.e., sign of the true value)
- an exponent within the specified word size (the **x** in 2^x), stored in either 2's complement or bias format
- a mantissa within the specified word size that stores the value of the *true value*

General format: for a word size of ***n***-bits (sign bit + exponent bits + mantissa bits = ***n*** bits):

```
   _ _______________ __________________
  |_|_______________|__________________|
sign   exponent        mantissa
```

- sign bit: represents the sign of the mantissa, using **sign-magnitude** form. Unlike integer and fixed-point that use 2's comp for negation, the floating-point sign bit is manipulated by the *floating-point routines* in the CPU or FPU: 0-positive, 1-negative.
- exponent: represents the original "unit position" of the *point*. The exponent is a power of 2 that indicates where the original point was. Surprisingly, there is more than one way to store the exponent value. The most *obvious* (and simplest) method is in 2's complement to accommodate negative and positive exponents. But the most *common* method in floating-point processing is *excess-n* (or *biased*) representation.

  > ***Note****: Biased exponent representation stores the value of the exponent as a positive integer in the range* ***0 to 2x****, whereas 2's complement represents the range* ***-(x+1) to x****. This is similar to comparing an* unsigned integer *(for biased) to a* signed integer *(for 2's complement), the range of values is the same with only the store value being different. Bias form is more efficient than 2's comp., introducing less "calculation overhead" during floating-point calculations.*
  > *Biased notation is left for another course or the student's personal interest.*

- mantissa: describes the actual value of the true value being represented, but in a full fractional form. The mantissa is defined by taking the fixed-point form of the true value and *normalising* it. The mantissa is always stored as the **positive** form of the number (the sign bit defines the true sign)
  most importantly: **except for a true value of zero, the mantissa must always begin with a "1"**
- overflow errors occur if the magnitude of a result is too large (exceeds the range of the exponent);
  underflow errors occur if the precision of a result exceeds the mantissa's word size (this always happens)

ex: (16-bit word, 7-bit mantissa, 8-bit exponent, 1-bit sign)

```
fixed-point:    10.25₁₀ = 1010.01₂
floating-point: 0 00000100 1010010
                0                  - sign bit
                  00000100         - exponent, 2's comp.
                           1010010  - mantissa
```

The most puzzling aspect of floating-point is the difficulty in determining the true value being represented. Yet considering that floating-point was never designed for a human to read or calculate with, its efficiency within the processor more than makes up for its strange and confusing look.

Normalisation

In order to represent a number in floating-point, we begin with the fixed-point form.

Normalise the fixed-point number above: 1010.01_2, the point is moved such that it is before the left-most 1 (msb):

```
1010.01 = .101001  (4 bits to the left)
```

Now that the number has been normalised, it can be stored in the mantissa: **1010010** (trailing zero added)

> ***Note****: When normalising, the fixed-point for is given unlimited magnitude and precision; over- and underflow will occur only during representation to floating-point.*

Exponent

From normalising the mantissa, the exponent is now known: $\mathbf{4}_{10}$ (because the "point" moved 4 bits to the left)

The exponent is represented in binary within the defined word size and in 2's complement form. In this case, the exponent is: $\mathbf{00000100}_2$

Putting it Together

Finally, the sign bit is recalled: 0-if true value positive, 1-if true value negative (sign bit can be defined any time).

All aspects are now ready, and the floating-number is assembled: **0 0000100 1010010** (written with spaces to assist in seeing the 3 aspects: *sign*, *exponent*, and *mantissa*).

Comments

- whether starting in decimal, or from a value already in binary, the fixed-point form must be positive before normalising to the mantissa; the sign of the true value is represented by the sign bit (0-negative, 1-positive)
- 2's complement is used only when storing the exponent, which is why *biased-form* has become popular, since it removes any negative in floating-point numbers (except for the sign bit, which isn't really negation)
- always be aware of the total word-size, and size for exponent and mantissa; these sizes constrain the range of possible values that can be represented by the floating-point data type (review Lecture 5)
- to convert from floating-point back to decimal, reverse the process that produces the fp representation
- most importantly: **floating-point is just a representation, not an actual binary number!**
  which implies that there must exist special routines to deal with floating-point values. [inside the NPU]

Self-Test Exercises

Convert the following numbers to floating-point form.

1. Represent the following fixed-point numbers in a floating-point of 14-bit word, 5-bit exponent, 8-bit mantissa. (the original fixed-point is an 9-bit word, 4-bit prec, 2's comp)

   a) 00101.0100_2 b) 11010.1100_2 (convert to positive first)

   c) 00000.0011_2 d) 11111.0011_2 (convert to positive first)

2. For a) and d) above, reverse the process and determine the decimal values from the floating-point results. Any problems?

3. Consider an FP with 12-bit word, 7-bit exponent, 4-bit mantissa. In decimal, what is the,

   a) "largest" number? (0 0111111 1111) ? b) "smallest" number (0 1000000 1000) ?

   c) [trick question] what are the "largest" negative and "smallest" negative numbers?

4. Convert the following decimal values to a floating-point with 16-bit word, 7-bit exponent, 8-bit mantissa.

   a) 245.453_{10} b) $-1.12 * 10^{-3}{}_{10}$

5. Describe how "overflow" and "underflow" result could be detected with the *floating-point* format.