§2.4 Floating Point

- floating-point representation encodes rational numbers of the form $V = x \times 2^y$
    - useful for performing computations involving...
        - very large numbers ($|V| >> 0$)
        - numbers very close to 0 ($|V| << 1$)
        - an approximation to real arithmetic
- up until the 1980s, every computer manufacturer devised its own conventions on how floating-point numbers were represented
    - speed & ease of implementation >>>> accuracy
- around 1985, **IEEE Standard 754** (standard for representing floating-point numbers & operations performed on them)
    - started in 1976 under Intel's sponsorship with the design of 8087
    - William Kahan (Cal professor) played a major role
- nowadays, virtually all computers support what has become known as ***IEEE floating point***
    - greatly improved the portability of scientific application programs across different machines

¶2.4.1 Fractional Binary Numbers

- familiar decmial notation

  $$d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d{-}2 \cdots d_{-n}$$

  where

  $$d = \sum_{i=-n}^{m} 10^i \times d_i$$

  --> weighting of the digits is defined relative to the decmial point symbol (.)

  --> digits to the left are weighted by nonnegative powers of 10 (integral values), while digits to the right are weighted by negative powers of 10 (fractional values)

- Consider a **binary notation**

  $$b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n+1} b_{-n}$$

  where each binary digit, or bit, b_i ranges between 0 and is defined as

  $$b = \sum_{i=-n}^{m} 2^i \times b_i$$

--> symbol '.' now becomes a **binary point**

--> bits on the left are weighted by nonnegative powers of 2, while bits on the right are weighted by negative powers of 2

$0.111\cdots1_2$ represents numbers just below 1

--> for shorthand notation $1.0-\epsilon$

- assuming we only consdier **finite-length encodings**, fractional binary notation can only represent numbers that can be written $x \times 2^y$
    - other values can only be approximated
        - must apporximate with increasing accuracy by lengthening the binary representation

H5 ¶2.4.2 IEEE Floating-Point Representation

- IEEE floating-point standard represents a number in a form

$$V = (-1)^s \times M \times 2^E$$

s *(sign)*, determines whether the number is negative ($s=1$) or positive ($s=0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case

M *(significand)*, fractional binary number thatn ranges either between 1 and $2-\epsilon$ or between 0 and $1-\epsilon$

E *(exponent)*, weights the value by a (possibly negative) power of 2

- bit representation of a floating-point number is divided into three fields
    1. single sign bit s
    2. k-bit exponent field $exp = e_{k-1}\cdots e_1 e_0$ encodes the exponent E
    3. n-bit fraction field $frac = f_{n-1}\cdots f_1 f_0$ encodes the significand M
        - value also depends on whether or not exponent field equals 0

- Single precision

```
31   30       23 22                            0
------------------------------------------------
| s |    exp    |              frac            |
------------------------------------------------
```

- float in C
- single sign bit s (1) + exp k (8) + frac n (23) = 32 bits

- Double precision

```
63   62            52 51                        32
------------------------------------------------
| s |      exp      |      frac(51:32)         |
------------------------------------------------
```

```
31                                             0
------------------------------------------------
|                 frac (31:0)                  |
------------------------------------------------
```

- double in C
- single sign bit s (1) + exp k (11) + frac n (52) = 64 bits

- when **exp** is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double)
- exponent field is interpreted as representing a signed integer in **biased** form
  - $E = e - Bias$

where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$ and $Bias$ is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double precision)

--> yields exponent ranges from -126 to 127 for single precision and -1022 to 1023 for double precision

- the fraction field is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$

--> then the significand is defined to be $M = 1 + f$ (***implied leading 1***) --> getting an **additional bit of precision** ($\because$ can adjust the exponent E)

- when **exp** is all **zeros**
- exponent value $E = 1 - Bias$, significand value $M = f$ (fraction field without an implied leading 1)
- serve two purposes
  1. to provide a way to **represent numeric value 0** ($M = f = 0$)

     ($\because$ for normalized numbers we must always have $M \geq 1$)

     - with IEEE floating-point format, -0.0 and $+0.0$ are considered different in some ways and the same in others
  2. to represent numbers that are very close to 0.0
     - ***gradual underflow*** --> possible numeric values are spaced evenly near 0.0

- when **exp** is all **ones**
  1. when **frac** is all zeros --> resulting values represent either $+\infty$ **or** $-\infty$ **depending on s**
     - can represent results that **overflow** (multiplying two very lare numbers or dividing by zero)
  2. when **frac** is nonzero --> resulting value = NaN (**not a number**)
     - when the result cannot be given as a real number or as infinity ($\sqrt{-1}$ or $\infty - \infty$)
     - or to represent uninitialized data

¶2.4.3 Example Numbers (w/o tables with example numbers)

- denormalized numbers are clustered around 0
- two zeros are special cases of denormalized numbers
- representable numbers are not uniformly distributed

  --> denser nearer the origin
- smooth transition between the largest denormalized number and the smallest normalized number?

  --> $\because$ definition of E for denormalized value ($E = 1 - Bias$)
- **IEEE** format was designed so that **floating-point numbers could be sorted using an integer sorting routines**

- no floating-point operations required to perform comparisons

- **Some general propeties for a floating point representation with a k-bit exponent and an n-bit fraction**:
  - $+0.0$ always has a bit representation of all zeros
  - smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all zeros

    $M = f = 2^{-n}, \quad E = -2^{k-1} + 2$ => $V = 2^{-n-2^{k-1}+2}$
  - largest denormalized value has a bit representation consisting of an exponent field of all zeros and a fraction field of all ones

    $M = f = 1 - 2^{-n} = 1 - \epsilon, \quad E = -2^{-k-1} + 2$ => $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$

    --> slightly smaller than the smallest normalized value
  - smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all zeros

    $M = 1, \quad E = -2^{k-1} + 2$ => $V = 2^{-2^{k-1}+2}$
  - value 1.0 has a bit representation with all but the most signifant bit of the exponent field equal to 1 and all other bits equal to 0

    $M = 1, \quad E = 0$
  - largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1

    $f = 1 - 2^{-n}$ => $M = 2 - 2^{-n} = 2 - \epsilon, \quad E = 2^{k-1} - 1,$ =>
    $V = (2 - 2^{-n} \times 2^{2^{k-1}-1}) = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$

¶2.4.4 Rounding

- floating-point arithmetic can only approximate real arithmetic ($\because$ representation has limited range & precision)
  - for a value x, need a systematic method of finding the "closest" matching value x' that can be represented in the desired floating-point format

    ==> "**Rounding operation**"
- **IEEE floating-point** format defines **FOUR** different *rounding modes*
  1. **Round-to-even** (round-to-nearest): Default
     - attempts to find a closest match

- when the value is halfway between two possible results, ==> **rounds the number either upward or downward** s.t. **the least significant digit of the result is even**

  £1.50 ==> £2, £2.50 ==> £2

2. **Round-toward-zero**
   - rounds positive numbers downward and negative numbers upward ($\hat{x}$ s.t. $|\hat{x}| \leq |x|$)

     £1.50 ==> £1, £2.50 ==> £2, £-1.50 => £-1

3. **Round-down**
   - rounds both positive and negative numbers downward (x^- s.t. $x^- \leq x$)

     £1.50 ==> £1, £2.50 ==> £2, £-1.50 => £-2

4. **Round-up**
   - rounds both positive and negative numbers upward (x^+ s.t. $x \leq x^+$)

- **Round-to-even**...? why do they prefer even numbers?
  - to avoid statistical bias (round up 50%, round down 50%)

H5 ¶2.4.5 Floating-Point Operations

- suppose x, y are real numbers, and $\exists$ some operation $\odot$ defined over real numbers, then the computation should yield $Round(x \odot y)$ (result of applying rounding to the exact result of the real operation)
- **IEEE standards** - point ops specification is independent of any particular hardware or software realization

- Let's define $x +^f$ to be $Round(x + y)$
  - defined for all values of x and y (although it may yield infinity even when both x and y are real numbers ==> **overflow**)
  - **Commutative** $x +^f y = y +^f x \quad \forall x, y \in \Re$
  - **Not associative**

    e.g $(3.14 + 1e10) - 1e10 = 0.0$, while $3.14 + (1e10 - 1e10) = 3.14$
  - most values have **inverses** under floating-point addition

    $x +^f -x = 0$

    except infinities, and NaNs $+\infty - \infty = NaN,\ NaN +^f x = NaN \quad \forall x \in \Re$

    - **Monotonicity property**

      if $a \geq b$, then $x +^f a \geq x +^f b \ \ \forall a, b, x \in \Re \backslash NAN$

      --> not obeyed by unsigned or two's-complement addition

- Lack of associativity??
    - Compilers tend to avoid any optimizations

      ```
      x = a + b + c;
      y = b + c + d;


      // Compiler might be tempted to optimize...
      t = b + c;
      x = a + t;
      y = t + d;
      ```

      --> might yield a different value ($\because$ different association of addition operations)

- Let's define $x *^f y$ to be $Round(x \times y)$
- closed under multiplication (although possibly yielding infinity or NAN)
- **commutative**
- has a **multiplicative identity = 1.0**
- **NOT associative** ($\because$ possibility of overflow & loss of precision due to rounding)

  e.g. $(1e20 * 1e20) * 1e-20 = +\infty$, $1e20 * (1e20 * 1e-20) = 1e20$
- **DOES NOT distribute over addition**

  e.g. $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = NaN$
- **Monotonicity properties**
    - $a \geq b$ and $c \geq 0$ => $a *^f c \geq b *^f c$
    - $a \geq b$ and $c \leq 0$ => $a *^f c \leq b *^f c$

  ==> do not hold for unsigned or two's-complement multiplications
- guaranteed that $a *^f a \geq 0$, as long as $a \neq NaN$

H5 ¶2.4.6 Floating Point in C

- all versions of C provide **two different floating-point data types** -- *float* (single-precision) & *double* (double-precision)
- machines use **round-to-even rounding mode**
  - no standard methods to change the rounding mode or to get special values ($-0, +\infty, -\infty$, or NAN)
- **GCC** defines **INFINITY** (for $+\infty$) and **NAN** (for NaN) when the program includes

```
#define _GNU_SOURCE 1
#include <math.h>
```

- **CASTING RULES!**
  1. ***int->float*** : number cannot overflow, but may be rounded
  2. ***int or float->double*** : exact numeric value can be preserved

     ($\because$ *double* has both greater range (range of representable values))
  3. ***double->float*** : can overflow to $+\infty$ or $-\infty$ since range is smaller. Otherwise, may be rounded
  4. ***float or double->int*** : will be rounded toward zero and value may overflow