Floating Point Arithmetic

CS 365

Floating-Point

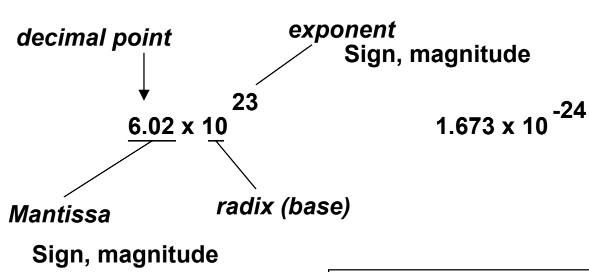
What can be represented in N bits?

- Unsigned 0 to 2^N
- 2s Complement -2^{N-1} to $2^{N-1} - 1$
- But, what about?
  - very large numbers?
    9,349,398,989,787,762,244,859,087,678
  - very small number?
    0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times \text{significand} \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

Recall Scientific Notation



IEEE F.P. $\pm 1.M \times 2^{e - 127}$

- Issues:
  - Arithmetic (+, -, *, /)
  - Representation, Normal form
  - Range and Precision
  - Rounding
  - Exceptions (e.g., divide by zero, overflow, underflow)
  - Errors

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+\text{significand}) \times 2^{exponent - bias}$
- Example:
  - decimal: $-.75 = -3/4 = -3/2^2$
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

IEEE 754 Standard

Representation of floating point numbers in IEEE 754 standard:

single precision

	1	8	23
sign	**S**	**E**	**M**
		exponent: **bias 127 binary integer**	***mantissa:*** **sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M**

actual exponent is
e = E - 127

0 < E < 255

$$N = (-1)^{S}\, 2^{E-127}\, (1.M)$$

0 = 0 00000000 0 . . . 0 **-1.5 = 1 01111111 10 . . . 0**

Magnitude of numbers that can be represented is in the range:

$$2^{-126}\,(1.0) \quad \text{to} \quad 2^{127}\,(2 - 2^{-23})$$

which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Floating Point Addition Example

<u>Example:</u> Add 9.999×10^1 and 1.610×10^{-1} assuming 4 decimal digits

1. Allign decimal point of number with smaller exponent

   $1.610 \times 10^{-1} = 0.161 \times 10^0 = 0.0161 \times 10^1$

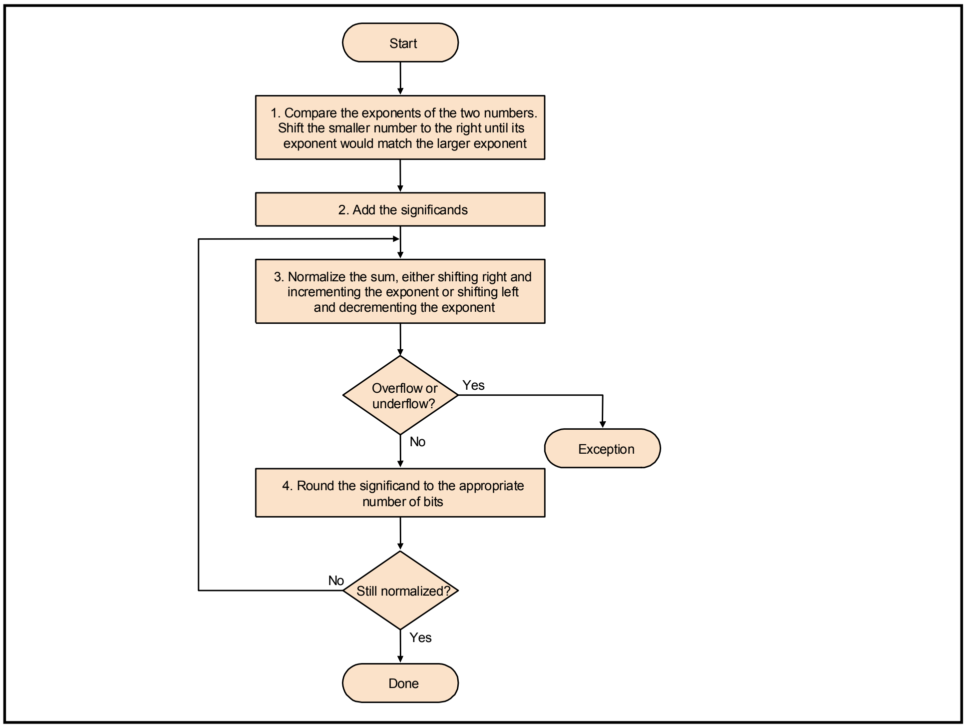
   Shift smaller number to right
2. Add significands

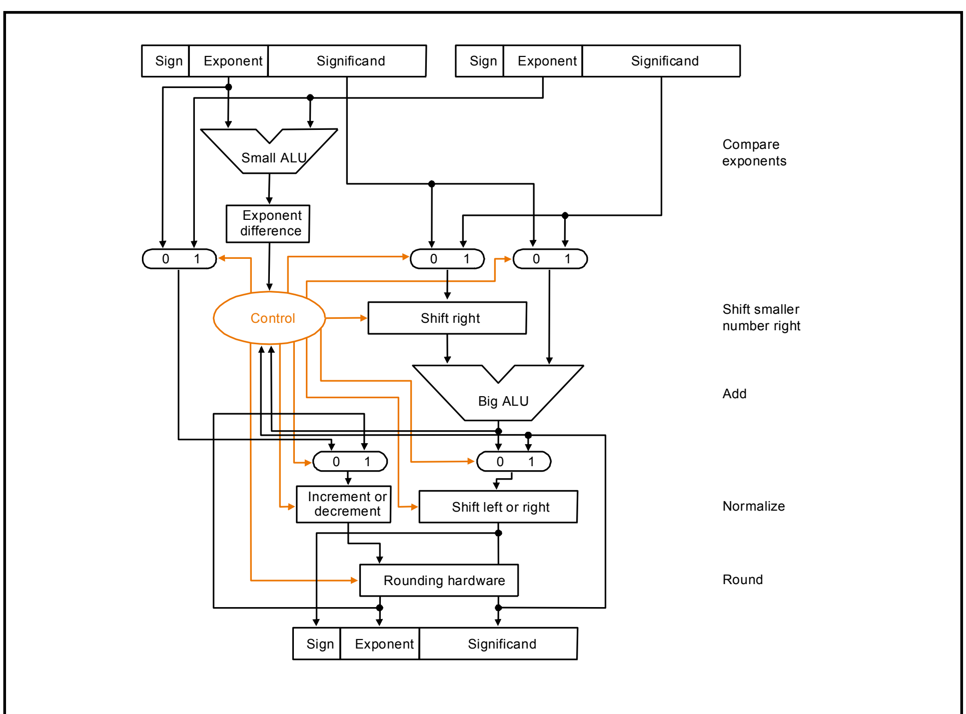
   9.999
   0.016
   10.015 ➔ SUM = 10.015×10^1

   NOTE: One digit of precision lost during shifting. Also sum is not normalized
3. Shift sum to put it in normalized form 1.0015×10^2
4. Since significand only has 4 digits, we need to round the sum

   SUM = 1.002×10^2

   NOTE: normalization maybe needed again after rounding,
   e.g, rounding 9.9999 you get 10.000


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done


Sign
Exponent
Significand
Sign
Exponent
Significand
Small ALU
Exponent difference
Control
Shift right
Big ALU
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Significand
Compare exponents
Shift smaller number right
Add
Normalize
Round

Accurate Arithmetic – Guard & Round bits

- IEEE 754 standard specifies the use of 2 extra bits on the right during intermediate calculations – Guard bit and Round bit
- Example: Add 2.56×10^0 and 2.34×10^2 assuming 3 significant digits and without guard and round bits

$$2.56 \times 10^0 = 0.0256 \times 10^2$$

$$\begin{array}{r} 2.34 \\ 0.02 \\ \hline 2.36 \times 10^2 \end{array}$$

- With guard and round bits

$$\begin{array}{r} 2.34 \\ 0.0256 \\ \hline 2.3656 \times 10^2 \end{array}$$

ROUND ➔ 2.37×10^0

Chapter Four Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).
- Next class: we are ready to move on (and implement the processor)