

Lecture 5

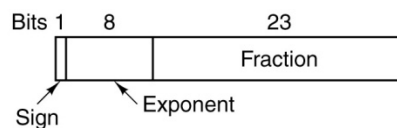
Floating Point Numbers

- Two's complement and floating point are the two standard number representations.
 - Floating point greatly simplifies working with large (e.g., 2^{70}) and small (e.g., 2^{-17}) numbers.
 - Early machines did it in software with “scaling factors”.
- We'll focus on the *IEEE 754* standard for floating-point arithmetic.
 - How FP numbers are represented.
 - Limitations of FP numbers.
 - FP addition and subtraction.

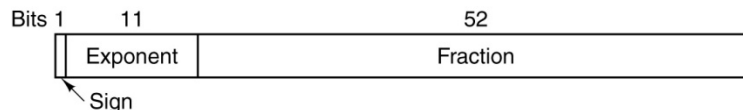
- IEEE* numbers are stored using a kind of scientific notation.

$$\pm \text{mantissa} * 2^{\text{exponent}}$$

- We can represent floating-point numbers with three binary fields: a sign bit s , an exponent field e , and a fraction field f .



(a)



- The *IEEE 754* standard defines several different precisions.
 - Single precision numbers consist of a 1-bit sign, an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.
 - Double precision numbers have a 1-bit sign, an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

- **Sign bit**

- The sign bit is 0 for positive numbers and 1 for negative numbers.
- But unlike integers, *IEEE* values are stored in signed magnitude format.

- **Exponent**

- The *e* field represents the exponent as a biased number.
 - It contains the actual exponent *plus* 127 for single precision, or the actual exponent *plus* 1023 in double precision.
 - This would convert all single-precision exponents from -127 to +128 into unsigned numbers from 0 to 255, and all double-precision exponents from -1023 to +1024 into unsigned numbers from 0 to 2047.
- Two examples with single-precision numbers are shown below.
 - If the exponent is 4, the *e* field will be $4 + 127 = 131$ (10000011_2).
 - If *e* contains 01011101 (93_{10}), the actual exponent is $93 - 127 = -34$.
- Storing a biased exponent *before* a normalized mantissa means we can compare *IEEE* values as if they were signed integers.

- **Mantissa/Fraction**

- The field *f* contains a binary fraction.
- The actual mantissa of the floating-point value is $(1 + f)$.
 - In other words, there is an implicit 1 to the left of the binary point.
 - For example, if *f* is $01101\dots$, the mantissa would be $1.01101\dots$
- There are many ways to write a number in scientific notation, but there is always a *unique* normalized representation, with exactly one non-zero digit to the left of the point.
 - $0.232 * 10^3 = 23.2 * 10^1 = 2.32 * 10^2 = \dots$
- A side effect is that we get a little more precision: there are 24 bits in the mantissa, but we only need to store 23 of them.

Conversion of IEEE floating point to decimal

- The decimal value of an IEEE number is given by the formula: $(1 - 2s) * (1 + f) * 2^{e - bias}$
- Here, the s , f and e fields are assumed to be in decimal.
 - $(1 - 2s)$ is 1 or -1, depending on whether the sign bit is 0 or 1.
 - We add an implicit 1 to the fraction field f , as mentioned earlier.
 - Again, the **bias** is either 127 or 1023, for single or double precision.

Conversion of Decimal to IEEE floating point

- What is the single-precision representation of 347.625?
 1. First convert the number to binary: $347.625 = 101011011.101_2$
 2. Normalize the number by shifting the binary point until there is a single 1 to the left:
 $101011011.101 \times 2^0 = 1.01011011101 \times 2^8$
 3. The bits to the right of the binary point comprise the fractional field f .
 4. The number of times you shifted gives the exponent. The field e should contain: **exponent + 127**
 5. Sign bit: 0 if positive, 1 if negative.

- Example: Find the decimal value of the following IEEE number.

1 01111100 110000000000000000000000

- Sign: negative
- Exponent: $\text{exponent} - \text{bias} = 124 - 127 = -3$
- Fraction: $1 + \text{fraction} = 1 + .75 = 1.75$
- Answer: $-1.75 * 2^{-3} = -0.21875$

Special values

- The smallest and largest possible exponents $e = 00000000$ and $e = 11111111$ (and their double precision counterparts) are reserved for special values.
- If the mantissa is always $(1 + f)$, then how is 0 represented?
 - The fraction field f should be 0000...0000.
 - The exponent field e contains the value 00000000.
 - With signed magnitude, there are *two* zeroes: +0.0 and -0.0.

- There are representations of positive and negative infinity, which might sometimes help with instances of overflow.
 - The fraction f is 0000...0000.
 - The exponent field e is set to 11111111.
- Finally, there is a special “*not a number*” (*nan*) value, which can handle some cases of errors or invalid operations such as 0.0/0.0.
 - The fraction field f is set to any non-zero value.
 - The exponent e will contain 11111111.

Normalized	±	0 < Exp < Max	Any bit pattern
Denormalized	±	0	Any nonzero bit pattern
Zero	±	0	0
Infinity	±	1 1 1...1	0
Not a number	±	1 1 1...1	Any nonzero bit pattern

↙ Sign bit

Range of single-precision numbers

What is the smallest positive single-precision value that can be represented?

$s = 0, e = 1 - bias, f = 0000...0$

$$(1 - 2s) * (1 + f) * 2^{e-127}$$

- And the smallest *positive* non-zero number is $1 * 2^{-126} = 2^{-126}$.
 - The smallest e is 00000001 (1).
 - The smallest f is 000000000000000000000000 (0).
- The largest possible “normal” number is $(2 - 2^{-23}) * 2^{127} = 2^{128} - 2^{104}$.
 - The largest possible e is 11111110 (254).
 - The largest possible f is 111111111111111111111111 (1 - 2⁻²³).
- In comparison, the range of possible 32-bit integers in two’s complement are only -2³¹ and (2³¹ - 1)

- How can we represent so many more values in the *IEEE 754* format, even though we use the same number of bits as regular integers?

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	-126 to +127	-1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Finiteness

- There *are not* more *IEEE* numbers.
- With 32 bits, there are 2^{32} , or about 4 billion, different bit patterns.
 - These can represent 4 billion integers *or* 4 billion reals.
 - But there are an infinite number of reals, and the *IEEE* format can only represent *some* of the ones from about -2^{128} to $+2^{128}$.
 - Represent same number of values between 2^n and 2^{n+1} as between 2^{n+1} and 2^{n+2} .
- Thus, floating-point arithmetic has “issues”.
 - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
 - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- The *IEEE 754* standard guarantees that all machines will produce the same results — but those results may not be mathematically correct!

Limits of the IEEE representation

- Even some integers cannot be represented in the IEEE format.

```
int   x = 33554431;
float y = 33554431;
printf( "%d\n", x );
printf( "%f\n", y );
```

```
33554431
33554432.000000
```

- Some simple decimal numbers cannot be represented exactly in binary to begin with.

$$0.1010_{10} = 0.0001100110011..._2$$

Floating-point addition

Example: $99.99 + 0.161 = 100.151$

- (In base 10, to get the idea) As normalized numbers, the operands would be written as:

$$9.999 * 10^1 \quad \text{and} \quad 1.610 * 10^{-1}$$

1. Equalize the exponents.

The operand with the smaller exponent should be rewritten by increasing its exponent and shifting the point leftwards.

$$1.610 * 10^{-1} = 0.0161 * 10^1$$

With four significant digits, this gets rounded to: $0.016 * 10^1$

This can result in a loss of least significant digits — the rightmost 1 in this case. But rewriting the number with the larger exponent could result in loss of the *most* significant digits, which is much worse.

2. Add the mantissas.

$$\begin{array}{r} 9.999 * 10^1 \\ + 0.016 * 10^1 \\ \hline 10.015 * 10^1 \end{array}$$

3. Normalize the result if necessary.

$$10.015 * 10^1 = 1.0015 * 10^2$$

This step may cause the point to shift either left or right, and the exponent to either increase or decrease.

4. Round the number if needed.

$1.0015 * 10^2$ gets rounded to $1.002 * 10^2$

5. Repeat Step 3 if the result is no longer normalized.

We do not need this in our example, but it is possible for rounding to add digits — for example, rounding 9.9995 yields 10.000.

Example 1: Add the following two numbers (in which the exponent is in excess 7 notation)

Variable	sign	exponent	fraction
<i>X</i>	0	1001	110
<i>Y</i>	0	0111	000

Here are the steps again:

1. **First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.**

In normalized scientific notation, *X* is 1.110×2^2 , and *Y* is 1.000×2^0 .

2. **In order to add, we need the exponents of the two numbers to be the same. We do this by rewriting *Y*. This will result in *Y* being not normalized, but value is equivalent to the normalized *Y*. Add $x - y$ to *Y*'s exponent (where x & y are the exponents of *X* & *Y*, respectively). Shift the radix point of the mantissa (significand) *Y* left by $x - y$ to compensate for the change in exponent.**

The difference of the exponent is 2. So, add 2 to *Y*'s exponent, and shift the radix point left by 2. This results in 0.0100×2^2 . This is still equivalent to the old value of *Y*. Call this readjusted value, *Y'*.

3. **Add the two mantissas of *X* and the adjusted *Y'* together.**

We add 1.110_2 to 0.01_2 . The sum is: 10.0_2 . The exponent is still the exponent of *X*, which is 2.

4. **If the sum in the previous step does not have a single bit of value 1, left of the radix point, then adjust the radix point and exponent until it does.**

In this case, the sum, 10.0_2 , has two bits left of the radix point. We need to move the radix point left by 1, and increase the exponent by 1 to compensate.

This results in: 1.000×2^3 .

5. **Convert back to the one byte floating point representation.**

Sum	sign	exponent	fraction
$X + Y$	0	1010	000

Example 2: Add the following two numbers

Variable	sign	exponent	fraction
X	0	1001	110
Y	0	0110	110

Here are the steps again:

1. **First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.**

In normalized scientific notation, X is 1.110×2^2 , and Y is 1.110×2^{-1} .

2. **In order to add, we need the exponents of the two numbers to be the same. We do this by rewriting Y . This will result in Y being not normalized, but value is equivalent to the normalized Y .**

Add $x - y$ to Y 's exponent. Shift the radix point of the mantissa ('significand') Y left by $x - y$ to compensate for the change in exponent.

The difference of the exponent is 3. So, add 3 to Y 's exponent, and shift the radix point of Y left by 3. This results in 0.00111×2^2 . This is still equivalent to the old value of Y . Call this readjusted value, Y' .

3. **Add the two mantissas of X and the adjusted Y' together.**

We add 1.110_2 to 0.00111_2 . The sum is: 1.11111_2 . The exponent is still the exponent of X , which is 2.

4. **If the sum in the previous step does not have a single bit of value 1, left of the radix point, then adjust the radix point and exponent until it does.**

In this case, the sum, 1.11111_2 , has a single 1 left of the radix point. So, the sum is normalized. We do not need to adjust anything yet.

So the result is the same as before: 1.11111×2^2 .

5. **Convert back to the one byte floating point representation.**

We only have 3 bits to represent the fraction. However, there were 5 bits in our answer. Obviously, it looks like we should round, and real floating point hardware would do rounding.

However, for simplicity, we're going to truncate the additional two bits. After truncating, we get 1.111×2^2 . We convert this back to floating point.

Sum	sign	exponent	fraction
$X + Y$	0	1001	111

This example illustrates what happens if the exponents are separated by too much. In fact, if the exponent differs by 4 or more, then effectively, you are adding 0 to the larger of the two numbers.

Floating Point Subtraction

- Like addition as far as alignment of radix points
- Use approach for sign magnitude subtraction:
 - If signs are different, then change the problem to addition:
 - For example, $(+24) - (-7) = +(24 + 7)$;
 - and $(-24) - (+7) = (-24) + (-7) = -(24 + 7)$
 - If signs are the same, then do subtraction:
 - Compare magnitudes, then subtract smaller from larger;
 - if the order is switched, then switch the sign too.
 - See the two examples below.

General rules of binary subtraction:

$$\begin{aligned}
 1 - 1 &= 0 \\
 0 - 0 &= 0 \\
 1 - 0 &= 1 \\
 10 - 1 &= 1 \\
 0 - 1 &= \text{borrow!}
 \end{aligned}$$

Example: sign magnitude subtraction:

$ \begin{array}{r} \underline{7 - 24} \\ 0\ 00111\ (7) \\ -\ 0\ 11000\ (24) \\ \hline \text{do} \\ \downarrow \\ 0\ 11000\ (24) \\ -\ 0\ 00111\ (7) \\ \hline 1\ 10001\ (-17) \end{array} $	$ \begin{array}{r} \underline{(-24) - (-2)} \\ 1\ 11000\ (-24) \\ -\ 1\ 00010\ (-2) \\ \hline 1\ 10110\ (-22) \end{array} $
--	---

(switch sign of the result since the order of the subtraction was reversed)

- Do not forget to normalize number afterward.