Real Numbers

We have been studying integer arithmetic up to this point.

We have discovered that a standard computer can represent a **finite subset** of the infinite set of integers. The range is determined by the number of bits used for the integers.

For example, the range for 16–bit two's complement arithmetic is –32,768 to 32,767.

We now turn our attention to **real numbers**, focusing on their representation
as **floating point numbers**.

The floating point representation of decimal numbers is often called **Scientific Notation**.

Most of us who use real numbers are more comfortable with **fixed point numbers**,
those with a fixed number of digits after the decimal point.

For example, normal U.S. money usage calls for two digits after the decimal: $123.45

Most banks use a variant of fixed point numbers to store cash balances and similar accounting data. This is due to the round–off issues with floating point numbers.

It might be possible to use 32–bit two's complement integers to represent the money in pennies. We could represent – $ 21,474,836.48 to $ 21,474,836.47

Floating Point Numbers

Floating point notation allows any number of digits to the right of the decimal point.

In considering decimal floating point notation, we focus on a standard representation, often called **"scientific notation"**, in which the number is represented as a product.

$$(-1)^S \bullet X \bullet 10^P, \text{ where } 1.0 \le X < 10.0$$

The restriction that $1.0 \le X < 10.0$ insures a unique representation.

Examples:	0.09375	$= (-1)^0 \bullet 9.375 \bullet 10^{-2}$
	– 23.375	$= (-1)^1 \bullet 2.3375 \bullet 10^1$
	1453.0	$= (-1)^0 \bullet 1.453 \bullet 10^3$
	$6.022142 \bullet 10^{23}$	Avogadro's Number, already in the standard form.

Avogadro's number, an experimentally determined value, shows 2 uses of the notation.

1. Without it, the number would require 24 digits to write.
2. It shows the precision with which the value of the constant is known. This says that the number is between $6.0221415 \bullet 10^{23}$ and $6.0221425 \bullet 10^{23}$.

QUESTION: What common number cannot be represented in this form?
HINT: Consider the constraint $1.0 \le X < 10.0$.

Zero Cannot Be Represented

In standard scientific notation, a zero is simply represented as 0.0.

One can also see numbers written as $0.0 \bullet 10^P$, for some power P, but this is usually the result of some computation and is usually rewritten as simply 0.0 or 0.00.

The constrained notation $(-1)^S \bullet X \bullet 10^P$ $(1.0 \leq X < 10.0)$, not normally a part of scientific notation, is the cause of the inability to represent the number 0.

Argument: Solve $X \bullet 10^P = 0$. Since $X > 0$, we can divide both sides by X.

We get: $10^P = 0$. But there is **no value P** such that $10^P = 0$.

Admittedly, $10^{-1000000}$ is so small as to be unimaginable, but it is not zero.

Having considered this non–standard variant of scientific notation, we move on and discuss **normalized binary numbers**.

Our discussion of floating point numbers will focus on a standard called

IEEE Floating–Point Standard 754, Single Precision

Normalized Binary Numbers

Normalized binary numbers are represented in the form.

$$(-1)^S \bullet X \bullet 2^P, \text{ where } 1.0 \le X < 2.0$$

Again, the constraint on X insures a unique representation. It also allows a protocol based on the fact that the first digit of the number X is always “1”.

In other words, X = 1.Y. Here are some examples.

$1.0 = 1.0 \bullet 2^0$, thus P = 0, X = 1.0 and, Y = 0.

$1.5 = 1.5 \bullet 2^0$, thus P = 0, X = 1.5 and, Y = 5.

$2.0 = 1.0 \bullet 2^1$, thus P = 1, X = 1.0 and, Y = 0.

$0.25 = 1.0 \bullet 2^{-2}$, thus P = –2, X = 1.0 and, Y = 0.

$7.0 = 1.75 \bullet 2^2$, thus P = 2, X = 1.75 and, Y = 75.

$0.75 = 1.5 \bullet 2^{-1}$, thus P = –1, X = 1.5 and, Y = 5.

The unusual representation of Y will be explained later.

The standard calls for representing a floating–point number with the triple (S, P, Y).

Representing the Exponent

The exponent is an integer. It can be either negative, zero, or positive.

In the IEEE Single Precision format, the exponent is stored as an 8–bit number
in excess–127 format.

Let P be the exponent. This format calls for storing the number (P + 127)
as an unsigned 8–bit integer.

The range of 8–bit unsigned integers is 0 to 255 inclusive. This leads to the following limits on the exponent that can be stored in this format.

$$0 \le (P + 127) \le 255$$

$$-127 \le P \le 128$$

Here are come examples.

P = – 5;	– 5 + 127 = 122.	Decimal 122 = 0111 1010 binary, the answer.
P = – 1;	– 1 + 127 = 126.	Decimal 126 = 0111 1110 binary, the answer.
P = 0;	0 + 127 = 127.	Decimal 127 = 0111 1111 binary, the answer.
P = 4;	4 + 127 = 131	Decimal 131 = 1000 0011 binary, the answer.
P = 33;	33 + 127 = 160	Decimal 160 = 1010 0000 binary, the answer.

IEEE Floating Point Standard 754 (Single Precision)

The standard calls for a 32–bit representation. From left to right, we have

One sign bit: 1 for a negative number and 0 for a non–negative number.

Eight exponent bits, storing the exponent in excess–127 notation.

23 bits for the **significand**, defined below.

The standard calls for two special patterns in the exponent field.

0000 0000	(P = – 127)	Reserved for **denormalized** numbers (not discussed)
1111 1111	(P = 128)	Reserved for infinity and NAN (Not a Number) Each defined below.

The range of exponents for a normalized number is – 127 < P < 128.

Normalized Numbers in IEEE Single Precision

In this standard, a normalized number is represented in the format:

$$(-1)^S \bullet X \bullet 2^P, \text{ where } 1.0 \le X < 2.0 \text{ and } -126 \le P \le 127.$$

The smallest positive number that can be represented as a normalized number in this format has value $\mathbf{1.0 \bullet 2^{-126}}$. We convert this to decimal.

$\text{Log}_{10}(2) = 0.301030$, so $\text{Log}_{10}(2^{-126}) = (-126) \bullet 0.301030 = -37.92978$
$= 0.07022 - 38$. But $10^{0.07022} \approx 1.2$.
We conclude that 2^{-126} is about $1.2 \bullet 10^{-38}$, the lower limit on this format.

The largest positive number that can be represented as a normalized number in this format has a value $(2 - 2^{-23}) \bullet 2^{127}$, minutely less than $2 \bullet 2^{127} = 2^{128}$.

Now $\text{Log}_{10}(2^{128}) = 128 \bullet 0.301030 = 38.53184$. Now $10^{0.53184} \approx 3.4$.
We conclude that 2^{128} is about $3.4 \bullet 10^{38}$, the upper limit on this format.

The range for positive normalized numbers in this format is $1.2 \bullet 10^{-38}$ to $3.4 \bullet 10^{38}$.

Infinity and NAN (Not A Number)

Here we speak loosely, in a fashion that would displease most pure mathematicians.

Infinity
What is the result of dividing a positive number by zero?

This is equivalent to solving the equation $X / 0 = Y$, or $0 \bullet Y = X > 0$, for some value Y.
There is no value Y such that $0 \bullet Y > 0$. Loosely we say that $X / 0 = \infty$.
The IEEE standard has a specific bit pattern for each ∞ and $-\infty$.

NAN
What is the result of dividing zero by zero?

This is equivalent to solving the equation $0 / 0 = Y$, or $0 \bullet Y = 0$.
This is true for every number Y. We say that $0 / 0$ is **not a number**.

It is easy to show that the mathematical monstrosities $\infty - \infty$ and ∞ / ∞
must each be set to NAN. This involves techniques normally associated with calculus.

An implementation of the standard can also use this "not a number" to represent other results that do not fit as real numbers. One example would be the square root of – 1.

Normalized Numbers: Producing the Binary Representation

Remember the structure of the single precision floating point number.

One sign bit: 1 for a negative number and 0 for a non–negative number.

Eight exponent bits, storing the exponent in excess–127 notation.

23 bits for the **significand**.

Step 1: Determine the sign bit. Save this for later.

Step 2: Convert the absolute value of the number to normalized form.

Step 3: Determine the eight–bit exponent field.

Step 4: Determine the 23–bit significand. There are shortcuts here.

Step 5: Arrange the fields in order.

Step 6: Rearrange the bits, grouping by fours from the left.

Step 7: Write the number as eight hexadecimal digits.

Exception: 0.0 is always 0x0000 0000. (Space used for legibility only)
This is a denormalized number, so the procedure does not apply.

Example: The Negative Number – 0.750

Step 1: The number is negative. The sign bit is S = 1.

Step 2: $0.750 = 1.5 \bullet 0.50 = 1.5 \bullet 2^{-1}$. The exponent is P = – 1.

Step 3: P + 127 = – 1 + 127 = 126. As an eight–bit number, this is 0111 1110.

Step 4: Convert 1.5 to binary. $1.5 = 1 + ½ = 1.1_2$. The significand is 10000.
To get the significand, drop the leading "1." from the number.
Note that we do not extend the significand to its full 23 bits, but
only place a few zeroes after the last 1 in the string.

Step 5: Arrange the bits: Sign | Exponent | Significand

```
Sign    Exponent   Significand
 1      0111 1110   1000 … 00
```

Step 6: Rearrange the bits

```
1011 1111 0100 0000 … etc.
```

Step 7: Write as 0xBF40. Extend to eight hex digits: **0xBF40 0000**.

The trick with the significand works because it comprises the bits to the right of the binary point. So, 10000 is the same as 1000 0000 0000 0000 0000 000.

Example: The Number 80.09375

This example will be worked in more detail, using methods that are more standard.

Step 1: The number is not negative. The sign bit is S = 0.

Step 2: We shall work this out in quite some detail, mostly to review the techniques.

Note that $2^6 \leq 80.09375 < 2^7$, so the exponent ought to be 6.

Convert 80.	80 / 2	= 40	remainder 0	
	40 / 2	= 20	remainder 0	
	20 / 2	= 10	remainder 0	
	10 / 2	= 5	remainder 0	
	5 / 2	= 2	remainder 1	
	2 / 2	= 1	remainder 0	
	1 / 2	= 0	remainder 1	101 0000 in binary.

Convert 0.09375	0.090375 • 2	= **0**.1875	
	0.1875 • 2	= **0**.375	
	0.375 • 2	= **0**.75	
	0.75 • 2	= **1**.50	(Drop the leading 1)
	0.50 • 2	= **1**.00	

The binary value is 101 0000.00011

Example: The Number 80.09375 (Continued)

Step 2 (Continued): We continue to convert the binary number 101 0000.00011.

To get a number in the form 1.Y, we move the binary point six places to the left. This moving six places to the left indicates that the exponent is P = 6.

101 0000.00011 = 1.0100 0000 011•2^6

Step 3: P + 127 = 6 + 127 = 133 = 128 + 5. In binary we have 1000 0101.

Step 4: The significand is 0100 0000 011 or 0100 0000 0110 0000.
Again, we just take the number 1.0100 0000 011 and drop the "1.".

Step 5: Arrange the bits: Sign | Exponent | Significand

```
Sign    Exponent          Significand
 0      1000 0101    0100 0000 0110 0000
```

Step 6: Rearrange the bits

```
0100 0010 1010 0000 0011 00000… etc.
```

Step 7: Write as 0x42A030. Extend to eight hex digits: **0x42A0 3000**.

Example in Reverse: 0x42E8 0000

Given the 32–bit number 0x42E8 0000, determine the value of the floating point number represented if the format is IEEE–754 Single Precision. Just do the steps backwards.

Step 1: From left to right, convert all non–zero hexadecimal digits to binary.
If necessary, pad out with trailing zeroes to get at least ten binary bits.

4	2	E	8
0100	0010	1110	1000

Step 2: Rearrange the bits as 1 bit | 8 bits | the rest

Sign	Exponent	Significand
0	1000 0101	1101000

Step 3: Interpret the sign bit. S = 0; the number is non–negative.

Step 4: Interpret the exponent field. $1000\ 0101_2 = 128 + 4 + 1 = 133$.
$P + 127 = 133$; $P = 6$.

Step 5: Extend and interpret the significand. Extend to 1.1101_2. Drop the trailing 0's.
$1.1101_2 = 1 + 1/2 + 1/4 + 1/16 = 1\ 13/16 = 1.8125$

Example in Reverse: 0x42E8 0000 (continued)

Step 6: Evaluate the number.
I show three ways to compute the magnitude.

6a Just do the multiplication.
We have $1.8125 \bullet 2^6 = 1.8125 \bullet 64 =$ **116.0**

6b Consider the fractional powers of 2. $1.1101_2 = 1 + 1/2 + 1/4 + 1/16$, so
we have $(1 + 1/2 + 1/4 + 1/16)\bullet 64 = 64 + 32 + 16 + 4 =$ **116.0**

6c The "binary" representation is $1.1101_2 \bullet 2^6$. Move the binary point
six places to the right to remove the exponent.
But first pad the right hand side of the significand to six bits.

The "binary" representation is $1.110100_2 \bullet 2^6$.
This equals 111 0100.0 = 64 + 32 + 16 + 4 = **116.0**

REMARK: Whenever the instructor gives more than one method to solve a problem, the student should feel free to select one and ignore the others.

Packed Decimal Arithmetic:
Another Representation of Real Numbers

Standard scientific computations seem to be best expressed using integer and standard floating–point arithmetic.

There is another format for representation of real numbers that is more appropriate for financial work.

This format is called **fixed point**, because the decimal point occupies a fixed point in the numeric representation.

Consider common financial transactions in dollar amounts. We usually denote this dollars and cents, with two digits to the right of the decimal place: $12.45.

Modern financial transactions may require more precision, but this can be satisfied by assuming more digits to the right of the decimal place: €50.123456.

Packed decimal arithmetic seems to have originated with early IBM computers, such as the IBM 702 and IBM 703, from 1953 to 1955.

The IA–32 series provides native support for packed decimal arithmetic, with the DAA and DAS instructions.

Precision Example: Slightly Exaggerated

Consider a banking problem. Banks lend each other money overnight.

At 3% annual interest, the overnight interest on \$1,000,000 is \$40.492.

Suppose my bank lends your bank \$10,000,000 (ten million).

You owe me \$404.92 in interest; \$10,000,404.92 in total.

With seven significant digits, the amount might be calculated as \$10,000,400.
My bank loses \$4.92.

I want my books to balance to the penny. I do not like floating point arithmetic.

TRUE STORY

When DEC (the Digital Equipment Corporation) was marketing their PDP–11
to a large New York bank, it supported integer and floating point arithmetic.

At this time, the PDP–11 did not support decimal arithmetic.

The bank told DEC something like this:
"Add decimal arithmetic and we shall buy a few thousand. Without it – no sale."

What do you think that DEC did?

Precision Example: Weather Modeling

Suppose a weather model that relies on three data to describe each point.

1. The temperature in Kelvins. Possible values are 200 K to 350 K.
2. The pressure in millibars. Typical values are around 1000.
3. The percent humidity. Possible ranges are 0.00 through 1.00 (100%).

Consider the errors associated with single precision floating point arithmetic.
The values are precise to 1 part in 10^7.

The maximum temperature errors resulting at any step in the calculation would be:

$3.5 \bullet 10^{-5}$ Kelvins

$1.0 \bullet 10^{-4}$ millibars

$1.0 \bullet 10^{-5}$ percent in humidity.

As cumulative errors tend to cancel each other out, it is hard to imagine
the cumulated error in the computation of any of these values
as becoming significant to the result.

Example: A weather prediction accurate to ±1 Kelvin is considered excellent.

Encoding Decimal Digits

There are ten decimal digits. As $2^3 < 10 \le 2^4$, we require four binary bits in order to represent each of the digits.

Here are the standard encodings. Note the resemblance to hexadecimal, except that the non–decimal hexadecimal digits are not shown.

0	0000	**5**	0101
1	0001	**6**	0110
2	0010	**7**	0111
3	0011	**8**	1000
4	0100	**9**	1001

Note that the binary codes	1010,	1011,	1100,	1101,	1110,	1111
for hexadecimal digits	A	B	C	D	E	F

are not used to represent decimal digits.

Packed decimal representation will have other uses for these binary codes.

Packed Decimal Data

Packed decimal representation makes use of the fact that only four binary bits are required to represent any decimal digit.

Numbers can be **packed**, two digits to a byte.

How do we represent signed numbers in this representation?
The answer is that we must include a sign "half byte".

The IBM format for packed decimal allows for an arbitrary number of digits in each number stored. The range is from 1 to 31, inclusive.

After adding the sign "half byte", the number of hexadecimal digits used to represent any number ranges from 2 through 32.

Numbers with an odd digit count are converted to proper format by the addition of a leading zero; "1234" becomes "01234", and "8" remains "8".

The system must allocate an integral number of bytes to store each datum, so each number stored in the format must have an odd number of digits.

Packed Decimal: The Sign “Half Byte”

In the S/370 Packed Decimal format, the sign is stored “to the right” of the string of digits as the least significant hexadecimal digit.

The standard calls for use of all six non–decimal hexadecimal digits as sign digits. The standard is as follows:

Binary	Hex	Sign	Comments
1010	A	+	
1011	B	–	
1100	C	+	The standard plus sign
1101	D	–	The standard minus sign
1110	E	+	
1111	F	+	A common plus sign, resulting from a shortcut in translation from Zoned format.

Packed Decimal Data

The packed decimal format is the one preferred by business for financial use.
Zoned decimal format is not used for arithmetic, but just for conversions.

As is suggested by the name, the packed format is more compact.

Zoned format	one digit per byte
Packed format	two digits per byte (mostly)

In the packed format, the rightmost byte stores the sign in its rightmost part, so the rightmost byte of packed format data contains only one digit.

All other bytes in the packed format contain two digits, each with value in 0 – 9.
This implies that each packed constants always has an odd number of digits.

A leading 0 may be inserted, as needed.

The standard sign fields are:	negative	X’D’
	non–negative	X’C’

The length may be from 1 to 16 bytes, or 1 to 31 decimal digits.

```
Examples    +7     | 7C |
            – 13   | 01 | 3D |
```

Example: Addition of Two Packed Decimal Values

Consider two **integer values**, stored in packed decimal format.

Note that 32–bit two's complement representation would limit the integer to a bit more than nine digits: –2, 147, 483, 648 through 2, 147, 483, 647.

Integers stored as packed decimals can have 31 digits and vary between
–9, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999 and
+9, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999 $(9.99 \bullet 10^{30})$

Consider the addition of the numbers –97 and 12,541.

–97 would be expanded to –097 and stored as **`097D`**.

12,541 would be stored as **`12541C`**.

The CPU does what we would do. It first pads the smaller number with 0's.

```
00097D
12541C
```

It then performs the operation, denoted as "12541 – 97", and stores the result as 12444, or **`12444C`** in packed decimal format.

Fixed Point: Where is the Decimal Point?

Consider the following addition problem: 1.23 + 10.11405.

The answer is obviously 11.34405.

But the Packed Decimal format does not store the decimal point, so

1.23 is stored as **`123C`** and 10.11405 is stored as **`1011405C`**.

Using our previous logic, we line up the numbers

```
0000123C
1011405C
```

The result is denoted **`10111528C`**, which might be 10.111528.
It cannot represent the correct answer.

This is a "feature" of **fixed point arithmetic**, in which all numbers are stored with the decimal point in the same place. In this system, the code must guarantee that 1.23 is stored as 0123000C.

The result is now

```
0123000C
1011405C,
```

stored as 1134405C.

The code must interpret and output this as the value 11.34405.