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.

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

$$-127 \leq P \leq 128$$

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 (defined below)
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 $3.4 \bullet 10^{38}$.

Denormalized Numbers

Consider two numbers, one small and one large. Each is a positive number that can be represented as a normalized number in this format.

Let $X = 10^{-20}$ and $Y = 10^{20}$.

Then $X / Y = 10^{-40}$, a number that cannot be represented in normalized form.
This leads to what is called an **underflow error**.

There are two options: either say that $X / Y = 0.0$ or store the number in another format.

The designers of the IEEE Floating Point Format devised **denormalized numbers** to handle this underflow problem. These are numbers with magnitude too small to be represented as normalized numbers.

The one very important denormalized number that is the exception here is **zero**.

Zero, denoted as "0.0", is the only denormalized number that will concern us.

The standard representation of 0.0 is just thirty two 0 bits.

0000 0000 0000 0000 0000 0000 0000 0000

0x00000000

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 $2^6 \le 80.09375 < 2^7$, 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	0.90375 • 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 = $1.0100\ 0000\ 011 \bullet 2^{6}$

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.

Example in Reverse: 0xC2E8 0000

This is a example rigged to make a particular point.

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.

C	2	E	8
1100	0010	1110	1000

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

Sign	Exponent	Significand
1	1000 0101	1101000

Here, we take a shortcut that should be obvious. Compare this bit pattern with that of the previous example, which evaluated to 116.0.

This pattern	1	1000 0101	1101000
116.0	0	1000 0101	1101000

This is the same number, just with a different sign bit. The answer is the negative number – 116.0.

A Final Example: 0xC000 0000

Step 1: From left to right, convert all non–zero hexadecimal digits to binary.

C

1100

If necessary, pad out with trailing zeroes to get at least ten binary bits.
Just to be thorough, I pad the number out to twelve binary bits.

C	0	0
1100	0000	0000

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

Sign	Exponent	Significand
1	1000 0000	0000

Step 3: Interpret the sign bit. S = 1; the number is negative.

Step 4: Interpret the exponent field. $1000\ 0000_2 = 128$.
P + 127 = 128; P = 1.

Step 5: Extend and interpret the significand. Extend to 1.0000_2. This is exactly 1.0

Step 6: Evaluate the number: $1.0 \bullet 2^1 =$ **2.0**.

Precision

How accurate is this floating point format?

Recall again the bit counts for the various fields of the 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**.

It is the 23 bits for the significand that give rise to the precision.

With the leading 1 (that is not stored), we have 24 bits, thus accuracy to 1 part in 2^{24}.

$2^{24} = 2^4 \bullet 2^{20} = 16 \bullet 2^{20} = 16, 777, 216.$

1 part in 10, 000, 000 would imply seven significant digits. This is slightly better, so we can claim seven significant digits.

The IEEE double precision format extends the accuracy to more digits.

Bankers and other financial types prefer exact arithmetic, so use another format (BCD) for all of their real number (money) calculations.

IBM S/370 Floating Point Data

We now discuss the representation used by IBM on its mainframe computers: the Sytem/360, System/370, and subsequent mainframes.

All floating point formats are of the form (S, E, F) representing $(-1)^S \bullet B^E \bullet F$.
It is the triple (S, E, F) that is stored in memory.

S	the sign bit, 1 for negative and 0 for non–negative.
B	the base of the number system; one of 2, 10, or 16.
E	the exponent.
F	the fraction.

The IEEE–754 standard calls for a binary base.

The IBM 370 format uses base 16.

Each of the formats represents the numbers in normalized form.

For IBM 370 format, this implies that $0.0625 < F \leq 1.0$. Note $(1/16) = 0.0625$.

S/370 Floating Point: Storing the Exponent

The exponent is stored in excess–64 format as a 7–bit unsigned number.

This allows for both positive and negative exponents.

A 7–bit unsigned binary number can store values in the range [0, 127] inclusive.

The range of exponents is given by $0 \le (E + 64) \le 127$, or

$-64 \le E \le 63$.

The leftmost byte of the format stores both the sign and exponent.

Bits	0	1	2	3	4	5	6	7
Field	Sign	Exponent in Excess–64 format						

Examples

Negative number, Exponent = –8 E + 64 = 56 = 48 + 8 = X’38’ = B’011 1000’.

0	1	2	3	4	5	6	7
Sign	3			8			
1	0	1	1	1	0	0	0

The value stored in the leftmost byte is 1011 1000 or B8.

Converting Decimal to Hexadecimal

The first step in producing the IBM 370 floating point representation of a real number is to convert that number into hexadecimal format.

The process for conversion has two steps,
one each for the integer and fractional part.

Example: Represent 123.90625 to hexadecimal.

Conversion of the integer part is achieved by repeated division with remainders.

123 / 16	= 7	with remainder	11	X'B'
7 / 16	= 0	with remainder	7	X'7'.

Read bottom to top as X'7B'. Indeed 123 = 7•16 + 11 = 112 + 11.

Conversion of the fractional part is achieved by repeated multiplication.

0.90625 • 16 = 14.5 Remove the 14 (hexadecimal E)

0.5 • 16 = 8.0 Remove the 8.

The answer is read top to bottom as E8.

The answer is that 123.90625 in decimal is represented by X'7B.E8'.

Converting Decimal to IBM 370 Floating Point Format

The decimal number is 123.90625.

Its hexadecimal representation is 7B.E8.

Normalize this by moving the decimal point two places to the left.

The number is now $16^2 \bullet 0.7BE8$.

The sign is 0, as the number is not negative.

The exponent is 2, E + 64 = 66 = X’42’. The leftmost byte is X’42’.

The fraction is 7BE8.

The left part of the floating point data is 427BE8.

In single precision, this would be represented in four bytes as 42 78 E8 00.

S/370: Available Floating Point Formats

There are three available formats for representing floating point numbers.

Single precision	4 bytes	32 bits: 0 – 31
Double precision	8 bytes	64 bits: 0 – 63
Extended precision	16 bytes	128 bits; 0 – 127.

The standard representation of the fields is as follows.

Format	Sign bit	Exponent bits	Fraction bits
Single	0	1 – 7	8 – 31
Double	0	1 – 7	8 – 63
Extended	0	1 – 7	8 – 127

NOTE: Unlike the IEEE–754 format, greater precision is not accompanied by a greater range of exponents.

The precision of the format depends on the number of bits used for the fraction.

Single precision	24 bit fraction	1 part in 2^{24}	7 digits precision *
Double precision	56 bit fraction	1 part in 2^{56}	16 digits precision **

*2^{24} = 16,777,216 ** $2^{56} \approx (10^{0.30103})56 \approx 10^{16.86} \approx 7 \bullet 10^{16}$.