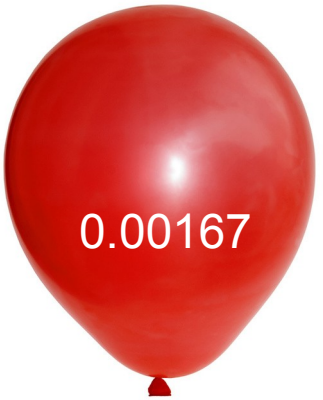
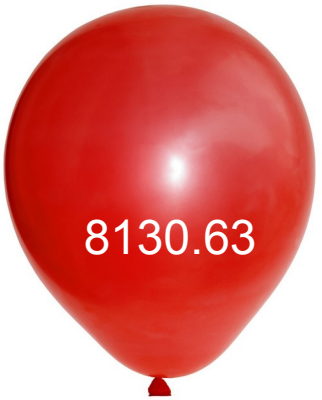
ENCE361



Phil Bones

Floating-point number representation



A one-off lecture supported by tutorial questions (and past exam questions)

“The purpose of computing is *insight*, not numbers”

Nevertheless, especially in engineering, we do need to represent numbers in the computer.

1. Number bases

In a position number system, a digit's value is determined by its position with respect to the decimal point or some other indicator. For example

$$347.2 = 3 \times 10^{2} + 4 \times 10^{1} + 7 \times 10^{0} + 2 \times 10^{-1}$$

The elements in a positional number system are:

the base *b*

the set of *b* symbols

the positional representation.

Hexadecimal is important in computing

$$1010001011111011002 = 28BEC16$$

1010001011111011002

1.1 Conversion between bases [page 2]

The integer and fractional parts are dealt with separately.

Consider converting the decimal number 61.18 into binary. For the **integer part**, use successive division by 2

					remainder
61	/	2	=	30	1
30	/	2	=	15	0
15	/	2	=	7	1
7	/	2	=	3	1
3	/	2	=	1	1
1	/	2	=	0	1

The process stops when the division produces a result of 0. The binary integer is just the remainders *read backwards*, thus

$61_{10} = 111101_2$

To convert the **fractional part**, use successive multiplication by 2

						integer	
2	x	0.18	=	0.36	+	0	
2	x	0.36	=	0.72	+	0	
2	x	0.72	=	0.44	+	1	
2	x	0.44	=	0.88	+	0	
2	x	0.88	=	0.76	+	1	
:	:	:	:	:		:	
2	x	0.24	=	0.48	+	0	
2	x	0.48	=	0.96	+	0	
2	x	0.96	=	0.92	+	1	
2	x	0.92	=	0.84	+	1,	and so on.

The integer portion of the product *read forwards* is the binary fraction. To 18 binary places of accuracy (using rounding) then,

$$0.18_{10} \approx 0.001011100001010010_2$$

and, combining the two parts,

$$61.18_{10} \approx 111101.001011100001010010_2$$

Ex 1. What does the following tiny C program print out? You should not need to actually compile and run the program to answer.

```
#include <stdio.h>

void main(void)
{
  float x = 0;

  while (x != 10)
    x = x + 0.1;
  printf("%f", x);
}
```

The code:

```
int n;
n = 1/3 + 1/3 + 1/3;
printf("%i", n);
```

The code:

```
int n;
n = 1/3 + 1/3 + 1/3;
printf("%i", n);
```

produces: 0

The code:

```
int n;
n = 1/3 + 1/3 + 1/3;
printf("%i", n);
```

produces: 0

The code:

```
float x;
x = 1/3 + 1/3 + 1/3;
printf("%f", x);
```

The code:

```
int n;
n = 1/3 + 1/3 + 1/3;
printf("%i", n);
```

produces: 0

The code:

```
float x;
x = 1/3 + 1/3 + 1/3;
printf("%f", x);
```

produces: 0.000000

Rounding in binary

110010.110101_2 *rounded* to 4 fractional bits is

Rounding in binary

110010.110101_2 *rounded* to 4 fractional bits is
110010.1101_2

110010.110110_2 *rounded* to 4 fractional bits is

Rounding in binary

110010.110101_2 *rounded* to 4 fractional bits is
110010.1101_2

110010.110110_2 *rounded* to 4 fractional bits is
110010.1110_2

Both numbers *truncated* to 4 fractional bits are:
110010.1101_2

2. Integer representations [page 4]

Unsigned numbers are frequently used for *counters* and for *addresses*.

With ***signed magnitude*** representation, the other *n* - 1 bits are used for the magnitude

S	magnitude	
1	*n* - 1	

The range of available numbers is then

$$-2^{n-1} + 1 \quad \text{to} \quad 2^{n-1} - 1$$

There are two representations for zero, namely **+0** and **-0**.

2's complement is the most universally employed signed integer representation in computers.

1 in the leftmost bit position (MSB) represents -2^{n-1}, while a 1 in the *i*th bit position, for *i* = 0, 1, 2, . . ., *n*-2, represents 2^i.

All bits 1 represents -1, and the range of numbers for 2's complement is

$$-2^{n-1} \quad \text{to} \quad 2^{n-1} - 1$$

Note that the MSB is still 1 for a negative number and 0 for a positive number, and there is now only a single representation for zero (all bits = 0).

binary	unsigned magnitude	signed magnitude	two's complement
00000000	0	0	0
00000001	1	1	1
00000010	2	2	2
00000011	3	3	3
:	:	:	:
01111110	126	126	126
01111111	127	127	127
10000000	128	0	-128
10000001	129	-1	-127
10000010	130	-2	-126
10000011	131	-3	-125
:	:	:	:
11111100	252	-124	-4
11111101	253	-125	-3
11111110	254	-126	-2
11111111	255	-127	-1

Table 3: 8-bit integers

Note: what a pattern of bits represents is entirely dependent on its role

3. Floating point representations

Floating point representations are used for numbers with fractional parts, i.e., real numbers. Real numbers are often used in engineering, and scientific notation is commonly used to represent these numbers. For example,

$$61.18_{10} = +0.6118 \times 10^{2}$$

or in general for base 10

$$y = S \times M \times 10^{E}$$

where ***S*** is the sign, ***M*** is the mantissa, and ***E*** is the exponent. The mantissa is usually *normalised*, which means that the exponent is adjusted so that the first non-zero digit appears immediately to the left or to the right of the decimal point. In computers base 2 is used, with the sign represented by the MSB bit as already discussed, so the general form is

$$y = (-1)^{S} \times M \times 2^{E}$$

For example consider the decimal number, **61.18**, which has the binary representation (from earlier slide):

$$\mathbf{61.18_{10} \approx 111101.00101110000101001 0_2}$$

To normalise, we move the point so that it is to the left of the first non-zero bit. This requires a move of 6 positions, so the exponent is $E = 6_{10} = 110_2$. The floating point binary number is then

$$\mathbf{61.18_{10} = +0.111101001011100001010010 \ldots \times 2^{110_2}}$$

Alternatively, the point can be moved so that it is to the right of the first bit, thus

$$\mathbf{61.18_{10} = +1.11101001011100001010010 \ldots \times 2^{101_2}}$$

The point of normalisation is to ensure that all bit positions in the mantissa are utilised, i.e. to maximise the number of significant digits (bits).

We have the problem of deciding how many bits to use altogether and how many to use for *M* and for *E*. As well, we know that there are choices to be made about the representations of *M* and *E* (e.g., *E* needs to be able to have negative as well as positive values).

3.1. The I.E.E.E. 754 floating point standard [page 6]

• Single precision (**`float`**) is a 32-bit format, with 1 sign bit, 8 exponent bits and 23 mantissa bits.
• Double precision (**`double`**) is a 64-bit format, with 1 sign bit, 11 exponent bits and 52 mantissa bits.

The second of the forms of normalisation is used for the representation of *most numbers*, i.e., the first bit (placed to the left of the point) is always 1. Because it is always 1, there is no need to store it (it can simply be assumed to be 1). That way the number of bits for the mantissa is effectively increased by one.

The number *zero* is treated as a special case.

The basic I.E.E.E. 754 single precision format is

$$y = (-1)^{S} \times 1.M \times 2^{E_{10} - 127}$$

so that the stored exponent value is biased (offset) by 127 to obtain the actual exponent. The "1.*M*" notation indicates that there is an *implied leading one* before the point with the stored mantissa, *M*, placed after the point.

The basic I.E.E.E. 754 double precision format is

$$y = (-1)^{S} \times 1.M \times 2^{E_{10} - 1023}$$

with the 11-bit exponent biased by 1023_{10}.

Example: [page 7]

The 4 bytes $BF08C000_{16}$ represent what number in I.E.E.E. 754 single precision format?

First expand and split into the three fields:

1011 1111 0000 1000 1100 0000 0000 0000

1	01111110	00010001100000000000000	
S	*E*	*M*	

Thus the number is:

$$(-1)^{1} \times 1.000100011 \times 2^{126 - 127} = -0.1000100011_{2}$$
$$= \mathbf{-0.5341796875_{10}}$$

3.1.1. Representing zero [page 7]

As mentioned above, I.E.E.E. 754 uses an implied leading one for representing numbers in its normalised format. Zero must therefore be treated separately. This is achieved by reserving the pattern with all bits of *E* = 0 for representing zero (although this idea is extended in 3.1.3, below), i.e.

X	**00000000**	**XXXXXXXXXXXXXXXXXXXXXXX**	**X = 0 or 1**
S	*E*	*M*	

3.1.2. Representing extreme numbers [page 7]

In engineering (and some other fields) there is often the need to represent extreme numbers, i.e. numbers which approach infinity. Another reserved pattern is used to represent "not a number" (***NaN***) for these cases, namely the pattern with all ones for *E*:

X	**11111111**	**XXXXXXXXXXXXXXXXXXXXXXX**	
S	*E*	*M*	

3.1.3. Denormals [pages 7 & 8]

It turns out that the normalised format selected for I.E.E.E. 754 does not cover the range of real numbers (the "number line") very evenly near zero.

Denormals are used to “fill the gap.”

Consider a *hypothetical* small version of I.E.E.E. 754 with only 8 bits:

1 for the sign, 3 for the exponent, and 4 for the mantissa.

This tiny system can only represent 128 magnitudes (i.e., excluding the sign), so we can form a table of all magnitudes (Table 4).

mantissa	exponent (E_2)							
(M)	000	001	010	011	100	101	110	111
0000	0.0	0.25	0.5	1.0	2.0	4.0	8.0	NaN
0001		0.265625	0.53125	1.0625	2.125	4.25	8.5	NaN
0010		0.28125	0.5625	1.125	2.25	4.5	9.0	NaN
0011		0.296875	0.59375	1.1875	2.375	4.75	9.5	NaN
0100		0.3125	0.625	1.25	2.5	5.0	10.0	NaN
0101		0.328125	0.65625	1.3125	2.625	5.25	10.5	NaN
:	:	:	:	:	:	:	:	:
1101		0.453125	0.90625	1.8125	3.625	7.25	14.5	NaN
1110		0.46875	0.9375	1.875	3.75	7.5	15.0	NaN
1111		0.484375	0.96875	1.9375	3.875	7.75	15.5	NaN

Table 4: Magnitudes represented by a hypothetical 8-bit version of I.E.E.E. 754.

Denormals, cont.

So we use the mantissa values other than 0000_2 associated with $E = 000_2$ to represent a set of numbers evenly spaced across the gap - *denormals*, so called because they are not normalised.

$$y_D = (-1)^S \times 0.M \times 2^{1 - 127_{10}} \qquad E = 0$$

for I.E.E.E. 754 single precision, or

$$y_D = (-1)^S \times 0.M \times 2^{1 - 3} \qquad E = 0$$

for the hypothetical 8-bit format above. The column of Table 4 for $E = 000_2$ now reads 0.0, 0.015625, 0.03125, 0.46875, etc.

The value zero is now represented by *all bits = 0*.

3.1.4. Maximum, minimum values [page 8]

The maximum and minimum magnitudes of numbers legitimately representable are:

precision	maximum magnitude	minimum magnitude
single	$2 \times 2^{127} = 3.403 \times 10^{38}$	$2^{-23} \times 2^{-126} = 1.401 \times 10^{-45}$
double	2×2^{1023} = huge!	$2^{-52} \times 2^{-1022}$ = tiny!

Table 5: Ranges for I.E.E.E. 754 floating point formats.

3.1.5. Summary

The I.E.E.E. 754 single precision format (**`float`**) is a 32-bit number, with 1 sign bit, 8 exponent bits and 23 mantissa bits. The bits are arranged:

S	*E*	*M*	

The value represented is:

$$y = (-1)^S \times 1.M \times 2^{E_{10} - 127} \qquad 0 < E < 255_{10}$$

$$y_D = (-1)^S \times 0.M \times 2^{1 - 127_{10}} \qquad E = 0 \text{ (denormals)}$$

$$NaN \qquad E = 255_{10}$$

4. Machine epsilon [page 9]

The traditional definition of *significant digits* ("figures") is:

The number of leading non-zero digits that are correct before the first incorrect digit is encountered.

$$\pi = 3.1415926\ \ldots$$

$$22 / 7 = 3.1428571\ \ldots$$

The *machine epsilon*, ε, is defined as the smallest possible increment in the mantissa that can be distinguished by the computer.

For a representation with an ***n***-bit mantissa and no leading 1, the machine epsilon is

$$\varepsilon = 2^{-n}$$

With an implied leading 1, the *effective* mantissa length is $n + 1$, so the machine epsilon is

$$\varepsilon = 2^{-(n+1)}$$

Consider the 8-bit floating point representation in Table 4 (with 4-bit mantissa and implied leading 1).

The machine epsilon is

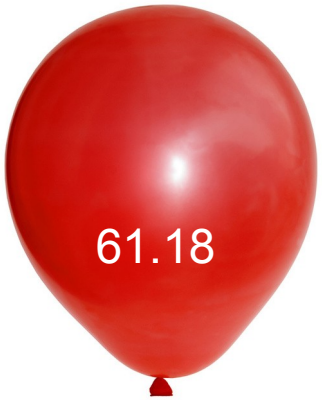
$$\varepsilon = 2^{-(4+1)} = .03125$$

From Table 4, the next largest number after 1.0 is 1.0625. With rounding, adding ε to 1.0 becomes

$$\mathbf{1.0 + 0.03125 = 1.0325 \approx 1.0625}$$

Adding anything smaller than 0.03125 rounds to 1.0.

The value of ε can therefore be determined for any machine by finding the smallest number which can be added to 1.0 such that the sum (as represented in the computer) is greater than 1.0.



---:---

ENEL323 2008 Examination Q2 (a)

Assume that the 4-byte pattern

80240000_{16}

is in IEEE 754 single precision format. Which number is represented? Express your answer in the form:

$a_{10} \times 2^n$