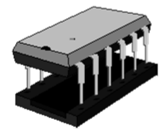
Question

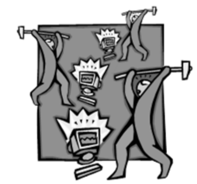
Do you have your laptop here?

A yes **B** no **C** what's a laptop **D** where is here?
E none of the above

Floating Point Numbers

6.626068 x 10^{-34}





- Eddie Edwards
- eedwards@doc.ic.ac.uk

- https://www.doc.ic.ac.uk/~eedwards/compsys
- Heavily based on notes from Naranker Dulay

Learning Outcomes

- At the end of this lecture you should
  - Understand the representation of real numbers in other bases (e.g 2)
  - Know the mantissa/exponent representation (in base 10, 2 etc.)
  - Be able to express numbers in normalised/un-normalised form
  - Be able to convert fractions/decimals between bases
  - Know the IEEE 754 floating point format (32 and 64 bit)
  - Know the special values and when they should occur
  - Understand the issues of accuracy in floating point representation

Number Representation - recap

- We have seen how to represent integers
  - positive integers as binary, octal and hexadecimal
  - negative integers as one's complement, two's complement, Excess-n
  - BCD, ASCI.....
- We have also seen how to perform arithmetic
  - Addition
    - by adding the binary bits
    - overflow conditions
  - Multiplication/division
    - same "long hand" techniques as base 10
    - slightly complicated in two's compliment
    - can take the absolute values, perform calculation, then sort out the sign

Numbers: Large, Small, Fractional

Population of the World	6,879,009,033 people
US National Debt (1990)	$3, 144, 830, 000, 000
1 Light Year	9, 130, 000, 000, 000 km
Mass of the Sun	2, 000, 000, 000, 000, 000, 000, 000, 000, 000 kg
Diameter of an Electron	0.000, 000, 000, 000, 000, 000, 01 m
Mass of an Electron	0.000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 9 kg
Smallest Measurable length of Time	0.000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 1 sec
Pi (to 8 decimal places)	3.14159265...
Standard Rate of VAT	17.5

Large Integers

Example: How can we represent integers up to 30 decimal digits long?

- **Binary** $\log_2 (10^{30})$ = ~ 100 bits (1 decimal digit = 3.322 bits)
- **BCD** 30 x 4-bit = 120 bits
- **ASCII** 30 x 8-bit = 240 bits

The Pentium includes instructions for writing multi-precision integer routines using Binary Coded Decimal (BCD) Arithmetic & ASCII arithmetic

Floating Pointing Numbers

Scientific Notation

Number = $M \times 10^E$ ← Decimal

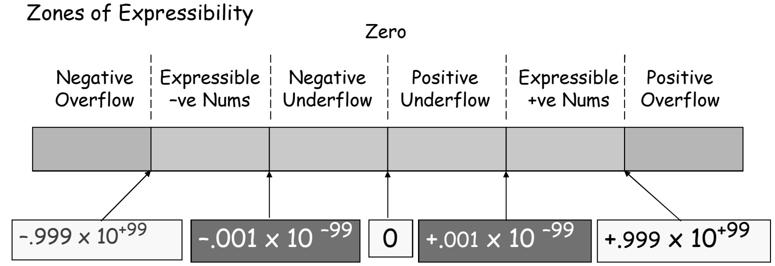
Number = $M \times 2^E$ ← Binary

- M is the **Mantissa** (or **Significand** or Fraction or Argument)
- E is the **Exponent** (or Characteristic)
- 10 (or for binary, 2) is the **Radix** (or **Base**)
- Digits (bits) in Exponent -> **Range** (Bigness/Smallness)
- Digits (bits) in Mantissa -> **Precision** (Exactness)

Zones of Expressibility

- **Example:** Assume numbers are formed with a Signed 3-digit Mantissa and a Signed 2-digit Exponent
- Numbers span from $\pm.001 \times 10^{-99}$ to $\pm.999 \times 10^{+99}$

Zones of Expressibility



Reals vs. Floating Point Numbers

	Mathematical Real	Floating-point Number
Range	-Infinity .. +Infinity	Finite
No. of Values	Infinite	Finite
Spacing	Constant & Infinite	Gap between numbers varies
Errors	?	Incorrect results are possible

Normalised Floating Point Numbers

- Floating Point Numbers can have multiple forms, e.g.

$$0.232 \times 10^{4} = 2.32 \times 10^{3} = 23.2 \times 10^{2} = 2\,320 \times 10^{0} = 232\,000 \times 10^{-2}$$

- For hardware implementation its desirable for each number to have a unique representation => **Normalised Form**
- We'll normalise Mantissa's in the **Range [1 .. R)** where R is the Base, e.g.:

  [1 .. 10) for DECIMAL
  [1 .. 2) for BINARY

Normalised Forms (Base 10)

Number	Normalised Form
23.2×10^{4}	2.32×10^{5}
-4.01×10^{-3}	-4.01×10^{-3}
$343\,000 \times 10$	3.43×10^{5}
$0.000\,000\,098\,9 \times 10^{0}$	9.89×10^{-8}

Binary & Decimal Fractions

Binary	Decimal
0.1	0.5
0.01	0.25
0.001	0.125
0.11	0.75
0.111	0.875
0.011	0.375
0.101	0.625

Binary Fraction to Decimal Fraction

- **Example**: What is the binary value **0.011010** in decimal ?

.	0	1	1	0	1
32	16	8	4	2	1

Sum = **8+4+1 = 13**

Answer: 13 / 32 = 0.40625

- Example: What is **0 . 0 0011 0011 00** in decimal ?

Answer: (32+16+2+1) / 512 = **51** / 512 = **0.099609375**

Decimal Fraction to Binary Fraction

- Example: What is 0.6875_{10} in binary ?

0.6875 * 2 = 1 .3750
0.3750 * 2 = 0 .7500
0.7500 * 2 = 1 .5000
0.5000 * 2 = 1 .0000
0.0000 * 2 = 0

Answer: 0.1011_2

- Example What is 0.1_{10} in binary ?

0.1_{10} in binary?

What is 0.1_{10} in binary ?

0.1 * 2 = 0 .2
0.2 * 2 = 0 .4
0.4 * 2 = 0 .8
0.8 * 2 = 1 .6
0.6 * 2 = 1 .2
0.2 * 2 = 0 .4 and then repeating 0.4, 0.8, 0.6

- Answer $0.0\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ \ldots\ _2$

Normalised Binary Floating Point Numbers

Number	Normalised Binary	Normalised Decimal
100.01×2^{1}	1.0001×2^{3}	8.5×10^{0}
1010.11×2^{2}	1.01011×2^{5}	4.3×10^{1}
0.00101×2^{-2}	1.01×2^{-5}	3.90625×10^{-2}
1100101×2^{-2}	$1.100101 \times 2^{+4}$	$9.86328125 \times 10^{-2}$

Floating Point Multiplication

$$N1 \times N2 = (M1 \times 10^{E1}) \times (M2 \times 10^{E2})$$
$$= (M1 \times M2) \times (10^{E1} \times 10^{E2})$$
$$= (M1 \times M2) \times (10^{E1+E2})$$

i.e. We multiply the Mantissas and Add the Exponents

Example: $20 * 6 = (2.0 \times 10^{1}) \times (6.0 \times 10^{0})$
$= (2.0 \times 6.0) \times (10^{1+0})$
$= 12.0 \times 10^{1}$

We must also normalise the result, so the final answer = 1.2×10^{2}

Truncation and Rounding

- For many computations the result of a floating point operation can be too large to store in the Mantissa.
- Example: with a 2-digit mantissa

  $2.3 \times 10^{1} * 2.3 \times 10^{1} = 5.29 \times 10^{2}$
- TRUNCATION => 5.2×10^{2} (Biased Error)
- ROUNDING => 5.3×10^{2} (Unbiased Error)

Floating Point Addition

- A floating point addition such as $4.5 \times 10^{3} + 6.7 \times 10^{2}$ is not a simple mantissa addition, unless the exponents are the same
  => we need to ensure that the mantissas are aligned first.

  $N1 + N2 = (M1 \times 10^{E1}) + (M2 \times 10^{E2})$

  $= (M1 + M2 \times 10^{E2-E1}) \times 10^{E1}$
- To align, choose the number with the smaller exponent & shift mantissa the corresponding number of digits to the right.

  Example: $4.5 \times 10^{3} + 6.7 \times 10^{2} = 4.5 \times 10^{3} + 0.67 \times 10^{3}$

  $= 5.17 \times 10^{3}$

  $= 5.2 \times 10^{3}$ (rounded)

Exponent Overflow & Underflow

- **EXPONENT OVERFLOW** occurs when the Result is too Large
  i.e. when the Result's Exponent > Maximum Exponent

  Example: if Max Exponent is 99 then $10^{99} * 10^{99} = 10^{198}$ (overflow)

  On Overflow => Proceed with incorrect value or infinity value or raise an Exception

- **EXPONENT UNDERFLOW** occurs when the Result is too Small
  i.e. when the Result's Exponent < Smallest Exponent

  Example: if Min Exp. is -99 then $10^{-99} * 10^{-99} = 10^{-198}$ (underflow)

  On Underflow => Proceed with zero value or raise an Exception

Comparing Floating-Point Values

- Because of the potential for producing in-exact results, comparing floating-point values should account for close results.
- If we know the likely magnitude and precision of results we can adjust for closeness (epsilon), for example, for equality we can:

  a = b   a > (b – e) AND a < (b + e)

  a = 1   a > (1 – 0.000005) AND a < 1 + 0.000005
  a >   0.999995 AND a < 1.000005

  Alternatively we can calculate | a - b | < e   e.g. | a - 1 | < 0.000005
- A more general approach is to calculate the closeness based on the relative size of the two numbers being compared.

Floating point numbers - questions

- What is the binary notation for 3.625
- A 11.011 B 10.101 C 11.101 D 101.11 E 11.11
- What is binary 0.1101 in decimal?
- A 0.8125 B 0.8 C 0.8625 D 0.9125 E 0.7865

IEEE Floating-Point Standard

- IEEE: Institute of Electrical & Electronic Engineers (USA)
- Comprehensive standard for Binary Floating-Point Arithmetic
- Widely adopted => Predictable results independent of architecture
- The standard defines:

  The format of binary floating-point numbers

  Semantics of arithmetic operations

  Rules for error conditions

Single Precision Format (32-bit)

Sign S	Exponent E	Significand F
1 bit	8 bits	23 bits

- The mantissa is called the **SIGNIFICAND** in the IEEE standard
- Value represented = $\pm 1.F \times 2^{E-127}$ $\quad 127 = 2^{8-1} - 1$
- The Normal Bit (the 1.) is omitted from the Significand field => a HIDDEN bit
- Single precision yields 24-bits = ~ 7 decimal digits of precision
- Normalised Ranges in decimal are approximately:

  -10^{+38} to -10^{-38}, **0,** $+10^{-38}$ to $+10^{+38}$

Exponent Field

- In the IEEE Standard, exponents are stored as Excess (Bias) Values, not as 2's Complement Values
- **Example:** In 8-bit Excess 127

  -127 would be held as 0000 0000
  ... ...
  0 would be held as 0111 1111
  1 would be held as 1000 0000
  ... ...
  128 would be held as 1111 1111

- Excess notation allows non-negative floating point numbers to be compared using simple integer comparisons, regardless of the absolute magnitude of the exponents.

Double Precision Format (64-bit)

Sign S	Exponent E	Significand F
1 bit	11 bits	52 bits

Value represented = $\pm 1.F \times 2^{E-1023}$ $\quad 1023 = 2^{11-1} - 1$

- Yields 53 bits of precision = ~ 16 decimal digits of precision
- Normalised Ranges in decimal are approximately:

  -10^{+308} to -10^{-308}, **0,** $+10^{-308}$ to $+10^{+308}$

- Double-Precision format is preferred for its greater precision. Single-precision is useful when memory is scarce and for debugging numerical calculations since rounding errors show up more quickly.

Example: Conversion to IEEE format

What is 42.6875 in IEEE Single Precision Format?

First convert to a binary number: **42.6875 = 10_1010 . 1011**

Next normalise: 1 . 0101_0101_1 x 2^5

Significand field is therefore: **0101_0101_1000_0000_0000_000**

Exponent field is (5+127=132): 1000_0100

Value in IEEE Single Precision is:

Sign	Exponent	Significand
0	1000_0100	**0101_0101_1000_0000_0000_000**

0100__0010__0 010__1010__1100__0000__0000__0000

In hexadecimal this value is **422A_C000**

Example: Conversion from IEEE format

Convert the IEEE Single Precision Value given by **BEC0_0000** to Decimal

BEC0_0000 = 1011_1110_1 100_0000_0000_0000_0000_0000

Sign	Exponent	Significand
1	**0111_1101**	**1000_0000_0000_0000_0000_000**

Exponent Field = 0111_1101 = 125
True Binary Exponent = 125 – 127 = –2

Significand Field = **1000_0000_0000_0000_0000_000**
Adding Hidden Bit = **1.1000_0000_0000_0000_0000_000**

Therefore unsigned value = 1.1 x 2^{-2} = 0 . 011 (binary)
= 0.25 + 0.125 = 0.375 (decimal)

Sign bit = 1 therefore number is -0.375

Example: Addition

- Carry out the addition 42.6875 + 0.375 in IEEE single precision arithmetic

Number	**Sign**	**Exponent**	**Significand**
42.6875	0	1000_0100	0101_0101_1000_0000_0000_000
0.375	0	0111_1101	1000_0000_0000_0000_0000_000

- To add these numbers the exponents of the numbers must be the same => Make the smaller exponent equal to the larger exponent, shifting the mantissa accordingly.
- Note: We must restore the Hidden bit when carrying out floating point operations.

Example: Addition Contd.

- Significand of Larger No = 1 . 0101_0101_1000_0000_0000_000
  Significand of Smaller No = 1 . 1000_0000_0000_0000_0000_000
- Exponents differ by +7 (1000_0100 – 0111_1101). Therefore shift binary point of smaller number 7 places to the left:
- Significand of Smaller No = 0 . 0000_0011_0000_0000_0000_000
  Significand of Larger No = 1 . 0101_0101_1000_0000_0000_000
  Significand of SUM = 1 . 0101_1000_1000_0000_0000_000
- Therefore SUM = 1 . **0101_1000_1** x 2^5 = 10_1011.0001 = 43.0625

Sign	Exponent	Significand
0	1000_0100	0101_1000_1 000_0000_0000_000 = 422C 4000H

Special Values

- The IEEE format can represent five kinds of values: Zero, Normalised Numbers, Denormalised Numbers, Infinity and Not-A-Numbers (NANs).
- For single precision format we have the following representations:

IEEE Value Field	Sign Field	Exponent Field	Significand Exponent	True
± Zero	0 or 1	0	0 (All zeroes)	
± Denormalised No	0 or 1	0	Any non-zero bit pat.	–126
± Normalised No	0 or 1	1 .. 254	Any bit pattern	-126 .. + 127
± Infinity	0 or 1	255	0 (All zeroes)	
Not-A-Number	0 or 1	255	Any non-zero bit pat.	

Denormalised Numbers

- An Exponent of All 0's is used to represent Zero and Denormalised numbers, while All 1's is used to represent Infinities and Not-A-Numbers (NaNs)
- This means that the maximum range for normalised numbers is reduced, i.e. for Single Precision the range is -126 .. +127 rather than -127 .. +128 as one might expect for Excess 127.

- **Denormalised Numbers** represent values between the Underflow limits and zero, i.e. for single precision we have:

$$\pm 0.F \times 2^{-126}$$

Traditionally a "flush-to-zero" is done when an underflow occurs

- Denormalised numbers allow a more gradual shift to zero, and are useful in a few numerical applications

IEEE 754 floating point numbers - questions

- What decimal is represented by the hex word C0CA0000

  Answer - -6.3125

- What hex word is -0.75 in IEEE-754?

  Answer - BFE8000000000000

Infinities and NaN's

- Infinities (both positive & negative) are used to represent values that exceed the overflow limits, and for operations like Divide by Zero
- Infinities behave as in Mathematics, e.g.

  Infinity + 5 = Infinity, -Infinity + -Infinity = -Infinity

- Not-A-Numbers (NaNs) are used to represent the results of operations which have no mathematical interpretation, e.g.

  0 / 0, +Infinity + -Infinity, 0 x Infinity, Square root of a -ve number,

- Operations with a NaN operand yield either a NaN result (quiet NaN operand) or an exception (signalling NaN operand)

This lecture - feedback

- The pace of the lecture was:

A. much too fast B. too fast C. about right D. too slow E. much too slow

- The learning objectives were met:

A. Fully B. Mostly C. Partially D. Slightly E. Not at all