COM1006 Devices and Networks (Autumn)
COM1090 Computer Architectures

Lecture #3

Computer arithmetic: Floating point numbers

Dr Dirk Sudholt
Department of Computer Science
University of Sheffield

d.sudholt@sheffield.ac.uk

http://staffwww.dcs.shef.ac.uk/~dirk/campus_only/com1006/

Based on Section 4.9 in Clements, Principles of Computer Hardware

Aims of this lecture

- To contrast fixed point and floating point arithmetic.
- To explain the terms *range*, *precision* and *accuracy* in relation to floating point numbers.
- To explain binary floating point representation.
- To show how floating point numbers can be represented in IEEE 754 floating point format.
- To briefly review floating point arithmetic.
- To discuss the implications of overflow and underflow in binary floating point calculations.

Dealing with fractional values

- In principle dealing with a decimal (or binary) fraction presents no problems.
- Consider the following calculations in decimal arithmetic:

```
 7632135          7632.135
+1794821         +1794.821
 -------          --------
 9426956          9426.956
```

integer arithmetic | **fractional arithmetic**

- Note that the calculations are identical apart from the position of the decimal point.
- We can do the same with computer arithmetic; the programmer remembers where the decimal point lies and all inputs and outputs are scaled accordingly.

Fixed point arithmetic

- This approach is called **fixed point arithmetic**; the binary point is assumed to remain in the same position.

  Example:

- Assume 8-bit fixed point numbers in which the 4 MSBs represent the integer part and the 4 LSBs represent the fractional part. Compute 3.625+6.5:

  $3.625_{10} \rightarrow 0011.1010_2$ (fractional part is 1/2+1/8 = 5/8)

  $6.5_{10} \rightarrow 0110.1000_2$ (fractional part is 1/2)

  Adding the binary numbers we get 10100010_2 which we interpret as $1010.0010_2 = 10.125$ (fractional part is 1/8).

Limitations of fixed point

- Fixed point arithmetic is used in certain applications:
  - Financial applications where control over rounding is required (e.g., ensure that smallest fractional part is 0.1p)
  - 3D graphics where speed is paramount or no special hardware is available. Sony Playstation uses fixed point.
- For many other applications fixed point arithmetic is unsuitable, because a large number of bits is required to represent numbers in a large range.
- **Example**: physicists need to handle numbers ranging from the mass of the sun (1.98892×10^{30} kg) to the mass of an electron ($9.10938188 \times 10^{-31}$ kg) which would require fixed point numbers hundreds of bits long.

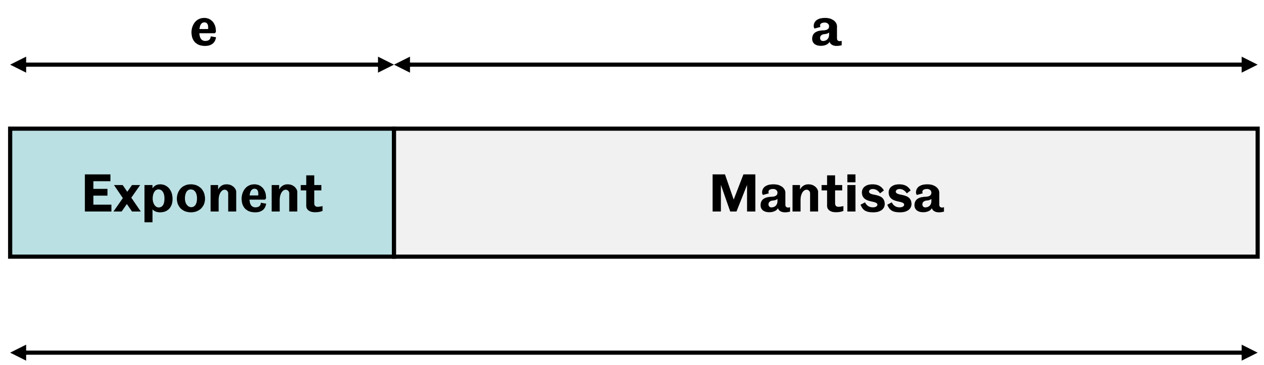
Representing floating point numbers

- In the last bullet point we used **scientific notation** or **floating point format**, i.e. we wrote 1.98892×10^{30} instead of 1988920000000000000000000000000.
- More generally, we represent floating point numbers in the form $a \times r^e$ where
  - a is the **mantissa** (or **argument**)
  - e is the **exponent**
  - r is the **radix** (or **base**)

$$\mathbf{a \times r^e}$$

- Computers store such numbers by splitting the mantissa and exponent into two fields. The radix is not explicitly stored (and from hereon is assumed to be 2).

Graphical representation of FP number



Floating point number $a \times 2^e$

- Must choose the following for a floating point representation:
  - the total number of bits used by the number
  - the number of bits allocated to the mantissa and exponent
  - representation of mantissa (two's complement etc.)
  - representation of exponent (biased etc. – see later)

Range and precision

- The **range** of a number determines how big or small it can be, e.g. in the example we had numbers between 2×10^{30} and 9×10^{-31}, a range of 10^{61}.
  - The more bits allocated to the **exponent**, the larger the **range**.
- The **precision** of a number is a measure of its exactness, and corresponds to the number of significant figures.
- Example: Pi may be written as 3.142 or 3.141592. The latter is more precise (it represents Pi to one part in 10^7, whereas the first is to one part in 10^4).
  - The more bits allocated to the **mantissa**, the larger the **precision.**
- **ANSI/IEEE 754 standard** offers good range and precision.

A word about accuracy

- The term **accuracy** is often used interchangeably with precision, but they have different meanings.
- Accuracy is a measure of **correctness** (how close an estimated value is to its true value).
- For example, 3.241592 is more **precise** than 3.141 because it has more significant digits, but it is less **accurate** because there is an error in the second digit.

▶Normalisation of IEEE FP numbers

- A floating point mantissa is **normalised** to the form 1.F where F is the fractional part, unless the mantissa is zero.

  Examples:

  $11.010\ldots_2 \times 2^{e}$ is normalised to $1.1010\ldots_2 \times 2^{e+1}$

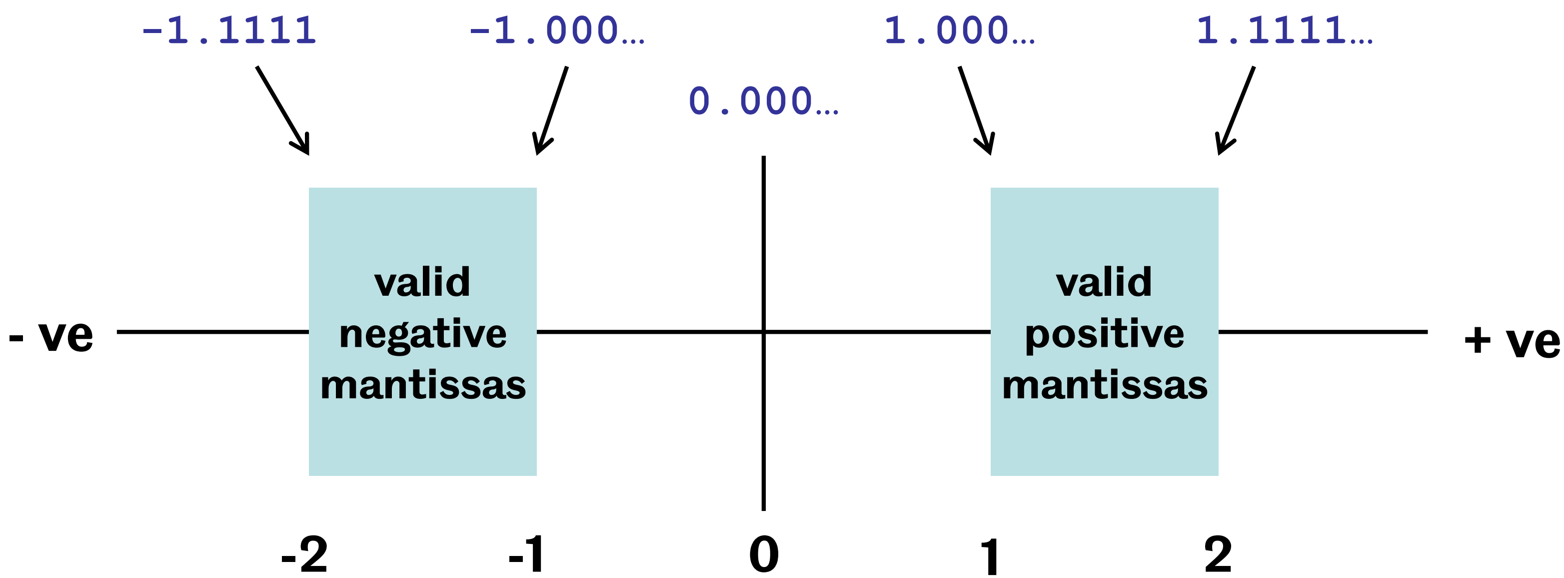
  $0.1001\ldots_2 \times 2^{e}$ is normalised to $1.001\ldots_2 \times 2^{e-1}$

❓ **What do we gain by this normalisation process?**

- IEEE floating point uses a sign and magnitude format; a **sign bit** indicates the sign of the mantissa.
- Positive mantissa is in the range $1.00\ldots_2$ to $1.11\ldots_2$, negative mantissa is in the range $-1.11\ldots_2$ to $-1.00\ldots_2$, i.e., the normalised mantissa *x* is limited to one of three ranges:

$$-2 < x <= -1 \quad \text{or} \quad x = 0 \quad \text{or} \quad 1 <= x < 2$$

Range of valid normalised mantissas



Representing the exponent

- We need to represent positive and negative exponents.
- In IEEE format the mantissa is represented in sign and magnitude form, but the exponent is in **biased** form.
- An *m*-bit exponent has 2^m unsigned values from 0 .. 2^m-1.
- We shift these values into the range $-2^{m-1}+1$ to $+2^{m-1}$ by subtracting a constant value (bias) $B = 2^{m-1}-1$ from each number.
- In other words: **we start counting from –B.**
- Example: Consider 3-bit biased exponent with $B=2^2-1=3$, the biased forms 0, 1, 2, 3, 4, 5, 6, 7 represent the true values -3, -2, -1, 0, 1, 2, 3, 4.
- A further example for m=4 and $B=2^3-1=7$ is shown next.

Example: biased exponents

- Consider the number `1010.1111`
- Normalized form is $+1.0101111 \times 2^3$
- True value is +3
- The value is actually stored in biased form which is $3+7 = 10_{10}$ or 1010_2 in binary form.

Binary value	True value	Biased form
0000	-7	0
0001	-6	1
0010	-5	2
0011	-4	3
0100	-3	4
0101	-2	5
0110	-1	6
0111	0	7
1000	1	8
1001	2	9
1010	3	10
1011	4	11
1100	5	12
1101	6	13
1110	7	14
1111	8	15

IEEE 754 floating point format

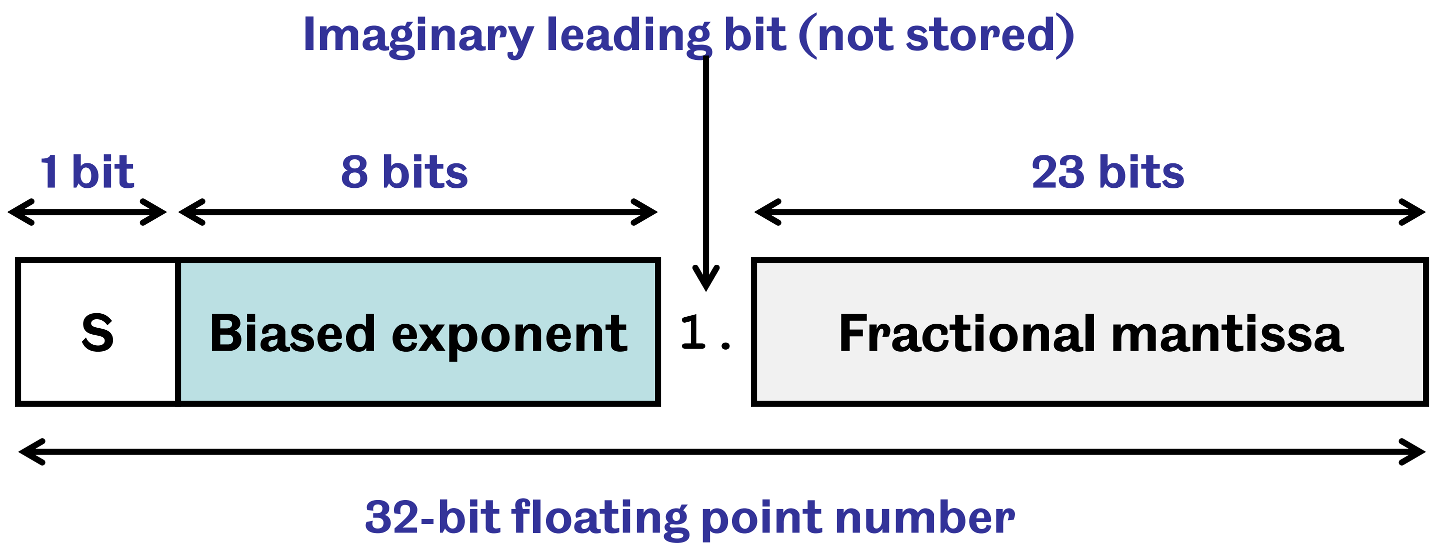
- An IEEE floating point number X is defined as

  $X = (-1)^{S} \times 2^{E-B} \times 1.F$

  where S = sign bit (0 = +ve mantissa, 1 = -ve mantissa),
  E = exponent biased by B, F = fractional mantissa.

- Note that mantissa is 1.F but only F is stored; the leading 1 is **implicit** ('imaginary leading bit'). By not storing it, the precision of the mantissa can be extended by one bit.

Format of IEEE 32-bit floating point



Bias for 8-bit exponent is $B=2^7-1=$**127**.

Decimal to IEEE FP: Example 1

What is the representation of the decimal number -2345.125 in IEEE 32-bit format?

- The equivalent binary number is -100100101001.001_2
- In normalised form this is $-1.00100101001001 \times 2^{11}$
- The mantissa is negative so the sign bit is 1.
- The bias for an IEEE 8-bit exponent is 127, so the biased exponent is given by 11 + 127 = 138 = 10001010_2
- The fractional part of the mantissa stored in 23 bits is 00100101001001000000000
- Hence -2345.125 is represented in IEEE 32-bit format as

  1 10001010 00100101001001000000000

Decimal to IEEE FP: Example 2

What is the representation of the decimal number 0.1875 in IEEE 32-bit format?

- The fractional part of the equivalent binary number is $0 \times 1/2 + 0 \times 1/4 + 1 \times 1/8 + 1 \times 1/16 = 0011_2$
- Equivalent binary number is 0.0011_2, sign bit is 0 (+ve)
- Normalised binary number is 1.1×2^{-3}
- Biased exponent is $-3 + 127 = 124 = 01111100_2$
- The fractional part of mantissa (stored in 23 bits) is `10000000000000000000000`
- Hence 0.1875 is represented in IEEE 32-bit format as

`0 01111100 10000000000000000000000`

Converting IEEE FP to decimal

- To convert IEEE 754 number to decimal, use the formula

$$(-1)^{S} \times (1+F) \times 2^{E-B}$$

- Example:

`0 01111100 10000000000000000000000`

- Sign bit S = `0`
- Fractional part of mantissa $F = 1000\ldots_2 = 1/2$
- Exponent $E = 01111100_2 = 124$
- So decimal form is $1 \times (1+0.5) \times 2^{124-127} = 1.5 \times 2^{-3} = 0.1875$

Summary of Integer representations

Word read as...	Unsigned integer	Sign & magnitude	Two's complement	Biased (B=7)
0000	0	0	0	-7
0001	1	1	1	-6
0010	2	2	2	-5
0011	3	3	3	-4
0100	4	4	4	-3
0101	5	5	5	-2
0110	6	6	6	-1
0111	7	7	7	0
1000	8	-0	-8	1
1001	9	-1	-7	2
1010	10	-2	-6	3
1011	11	-3	-5	4
1100	12	-4	-4	5
1101	13	-5	-3	6
1110	14	-6	-2	7
1111	15	-7	-1	8

IEEE floating point formats

- To cater for different applications, the IEEE 754 standard specifies three basic formats, called **single**, **double** and **quad**.
- For the assessment you only need to know the single precision format.

Basic IEEE floating point formats

	Single precision	Double precision	Quad precision
Field width in bits			
S = sign	1	1	1
E = exponent	8	11	15
L = leading bit	1	1	1
F = fraction	23	52	112
Total width in bits	32	64	128
Exponent			
Maximum E	$127 = 11...110_2$	$1023 = 11...110_2$	$16383 = 11...110_2$
Minimum E	$-126 = 00...001_2$	$-1022 = 00...001_2$	$-16382 = 00...001_2$
Bias	127	1023	16383

Representing zero, infinity and NaN

- In 32-bit single precision IEEE format exponents E_{min} -1 = -127 = 0000000_2 and E_{max} +1 = +128 = 11111111_2 have a special interpretation.

- The special value E_{min}-1 = -127 = $00...00_2$ is used to encode zero:

sign	exponent	mantissa
0	000 … 0	000 … 0

- Fits in neatly as decreasing exponents yield numbers closer to 0.

- E_{max}+1 = 128 is used to encode plus or minus infinity, or a **not a number** (NaN) resulting from division by zero.

▶Floating point arithmetic

- Consider the addition of two floating point numbers:

$$
\begin{array}{ll}
 & 1.110100 \times 2^5 \\
+ & 1.010001 \times 2^3
\end{array}
$$

- We can't add these numbers directly because the exponents are different. Instead we must:

1. Identify the number with the smaller exponent;
2. Make the smaller exponent equal to the larger exponent by dividing the mantissa of the smaller number by the same factor by which its exponent must be increased;
3. Add (or subtract) the mantissas;
4. If necessary, normalise the result (post-normalisation)
5. Truncate or round the mantissa.

Example: floating point addition

- Consider sum of A = 1.110100×2^5 and B = 1.010001×2^3 with 6-bit mantissas (and imaginary leading 1).
- The exponent of B is smaller than that of A, so divide B's mantissa by 2^2 (shift to the right by 2 bits) to give 0.01010001×2^5
- $$\begin{array}{rl} & 1.110100 \times 2^5 \\ + & 1.010001 \times 2^3 \end{array} \rightarrow \begin{array}{rl} & 1.110100 \quad\;\; \times 2^5 \\ + & \underline{0.01010001 \times 2^5} \\ & 10.00100001 \times 2^5 \end{array}$$

post-normalisation: 1.000100001×2^6

truncation/rounding: 1.000100×2^6

Implications of floating point

- Every time a calculation is done, further accuracy may be lost from the result due to **truncation** or **rounding**.
- Not all numbers can be represented: the number 0.75 can be represented exactly in binary floating point format

  $0.75_{10} = 0.11_2$

  however this is generally not the case, e.g.

  $0.1_{10} = 0.000110011001\ldots_2$

  (same problem in decimal: $1/3 = 0.33333333\ldots_{10}$)
- If X is much larger than Y and Y > 0, we might get X+Y=X.

Overflow and underflow

- During post-normalization the exponent is checked as follows:
- **Exponent overflow**: exponent is more than the maximum possible value. This is an error condition.
- **Exponent underflow**: exponent is less than the minimum possible value.
- A simple way of fixing underflow is to set the result to zero.
- IEEE 754 uses **gradual underflow**: very small numbers are represented as **denormalised** numbers without leading 1: if exponent is 00...00 numbers are $(-1)^{S} \times 2^{(-B+1)} \times 0.F$ so that the approach towards zero is more gradual.

sign	exponent		mantissa
0	000 … 0	0 .	000 … 01

smallest denormalised number: $2^{-23} \times 2^{-126} = 2^{-149}$

Summary

- Integers can be represented in different ways: unsigned integers, two's complement, sign & magnitude, biased representation.
- We can represent fractional values using fixed point or floating point binary representations. The latter is usually preferred, and the current standard is IEEE 754:
  - the mantissa is represented in normalised sign and magnitude form
  - the exponent is represented in biased form.
- Learned how to convert between decimal numbers and IEEE 754.
- Two floating point numbers are added by increasing the smaller exponent and adding, normalising, and truncating/rounding the mantissas.
- Underflow, overflow and rounding/truncation errors can affect the accuracy of floating point calculations.