Computer Representation of Numbers and Computer Arithmetic

January 21, 2019

Contents

1 Binary numbers **6**

2 Signed Integer Numbers **9**

2.1 Representing integer numbers in binary 9

2.2 Storing Integers in Memory 14

3 Floating-Point Numbers **17**
3.1 Scientific representation of real numbers 17
3.2 Floating-point: representation of real numbers on finite real estate 19
3.3 Roundoff (approximation) errors 20
3.4 Details of the floating point representation 22
3.5 Binary floating point numbers 26

4 The IEEE standard **28**
4.1 Floating Point Types 29
4.2 Detailed IEEE representation 31
4.3 Number range 31
4.4 Precision 34

5 The Set of Floating Point Numbers **41**

6 Rounding **43**
6.1 Rounding up or down 43
6.2 Rounding to zero (“chopping”) 45
6.3 Rounding to nearest 47

6.4 Summary of rounding modes 50

7 Arithmetic Operations 52

7.1 IEEE Arithmetic 56

8 Pitfalls with Floating Point Arithmetic 58

8.1 Binary versus decimal 58
8.2 Floating point comparisons 59
8.3 Funny conversions 61
8.4 Memory versus register operands 64
8.5 Cancellation ("Loss-of Significance") Errors 64
8.6 Insignificant Digits 67
8.7 Order matters 68

9 Integer Multiplication 70

10 Special Arithmetic Operations 71

10.1 Signed zeros 71
10.2 Operations with ∞ 72
10.3 Operations with `NaN` 72

10.4 Comparisons 73

11 Arithmetic Exceptions 74

11.1 Division by 0 74
11.2 Overflow 75
11.3 Underflow 77
11.4 Inexact 79
11.5 Summary 79
11.6 Example: messy mathematical operations 79
11.7 Example: overflow 81
11.8 Example: underflow 83

12 Long Summations 85

1 Binary numbers

In the decimal system, the number 107.625 means

$$107.625 = 1 \cdot 10^2 + 7 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3} \ .$$

Such a number is the sum of terms of the form {a digit times a different power of 10} - we say that 10 is the *basis* of the decimal system. There are 10 digits (0,...,9).

All computers today use the *binary system*. This has obvious hardware advantages, since the only digits in this system are 0 and 1. In the binary system the number is represented as the sum of terms of the form {a digit times a different power of 2}. For example,

$$\begin{aligned} (107.625)_{10} &= 2^6 + 2^5 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} \\ &= (1101011.101)_2 \ . \end{aligned}$$

Arithmetic operations in the binary system are performed similarly as in

the decimal system; since there are only 2 digits, 1+1=10.

$$
\begin{array}{rcccccc}
 & & 1 & 1 & 1 & 1 & 0 \\
+ & & & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\qquad
\begin{array}{rcccccc}
 & & & & 1 & 1 & 1 \\
\times & & & & 1 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 \\
 & & & 1 & 1 & 1 & \\
 & & 1 & 1 & 1 & & \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

Decimal to binary conversion. For the integer part, we divide by 2 repeatedly (using integer division); the remainders are the successive digits of the number in base 2, from least to most significant.

Quotients	107	53	26	13	6	3	1	0
Remainders		1	1	0	1	0	1	1

For the fractional part, multiply the number by 2; take away the integer part, and multiply the fractional part of the result by 2, and so on; the sequence of integer parts are the digits of the base 2 number, from most to least significant.

Fractional	0.625	0.25	0.5	0
Integer		1	0	1

Octal representation. A binary number can be easily represented in base 8. Partition the number into groups of 3 binary digits ($2^3 = 8$), from decimal point to the right and to the left (add zeros if needed). Then, replace each group by its octal equivalent.

$$(107.625)_{10} = (\boxed{1}\boxed{101}\boxed{011}.\boxed{101})_2 = (153.5)_8$$

Hexadecimal representation. To represent a binary number in base 16 proceed as above, but now partition the number into groups of 4 binary digits ($2^4 = 16$). The base 16 digits are 0,...,9,A=10,...,F=15.

$$(107.625)_{10} = (\boxed{0110}\boxed{1011}.\boxed{1010})_2 = (6B.A)_{16}$$

Homework 1 1. *Convert the following binary numbers to decimal, octal and hexa: 1001101101.0011, 11011.111001;*

2. *Convert the following hexadecimal numbers to both decimal and binary: 1AD.CF, D4E5.35A;*

3. *Convert the following decimal numbers to both binary and hexadecimal: 6752.8756, 4687.4231.*

2 Signed Integer Numbers

2.1 Representing integer numbers in binary

Unsigned binary integers.

- m binary digits (bits) of memory can store 2^m different numbers. They can be positive integers between $\boxed{00\ldots00} = (0)_{10}$ and $\boxed{11\ldots11} = (2^m - 1)_{10}$.

- For example, using $m = 3$ bits, we can represent any integer between 0 and 7.

In order to represent signed integers (i.e. both positive and negative numbers) using m bits, we can use one of the methods discussed next.

Sign/Magnitude representation.

- Reserve the first bit for the signum (for example, let 0 denote positive numbers, and 1 negative numbers).
- The other $m-1$ bits will store the magnitude (the absolute value) of the number.
- In this case the range of numbers represented is $-2^{m-1}+1$ to $+2^{m-1}-1$. With $m=3$ there are 2 bits for the magnitude, different possible magnitudes, between 0 and 127; each of these can have a positive and negative sign. Note that with this representation we have both positive and negative zero.
- If we make the convention that the sign bit is 1 for negative numbers we have:

Number_{10}	$([\text{S}]\text{M})_2$
-3	[1]11
-2	[1]10
-1	[1]01
-0	[1]00
+0	[0]00
+1	[0]01
+2	[0]10
+3	[0]11

Two's complement representation.

- All numbers from -2^{m-1} to $+2^{m-1}-1$ are represented by the smallest positive integer with which they are congruent modulo 2^m.

- With $m = 3$, for example, we have:

Number_{10}	$(2C)_{10}$	$(2C)_2$
-4	4	100
-3	5	101
-2	6	110
-1	7	111
0	0	000
1	1	001
2	2	010
3	3	011

- Note that the first bit is 1 for negative numbers, and 0 for nonnegative numbers.

Biased representation.

- A number $x \in [-2^{m-1}, 2^{m-1} - 1]$ is represented by the positive value $\bar{x} = x + 2^{m-1} \in [0, 2^m - 1]$.
- The mapping is done by adding the *bias* 2^{m-1} to obtain positive results.
- For $m = 3$ we have $bias = 4$ and the representation:

Number_{10}	$(\text{biased})_{10}$	$(\text{biased})_2$
-4	0	000
-3	1	001
-2	2	010
-1	3	011
0	4	100
1	5	101
2	6	110
3	7	111

- The first bit is 0 for negative numbers, and 1 for nonnegative numbers.

2.2 Storing Integers in Memory

One byte of memory can store $2^8 = 256$ different numbers. They can be positive integers between $\boxed{00000000} = (0)_{10}$ and $\boxed{11111111} = (255)_{10}$.

For most applications, one byte integers are too small. Standard data types usually reserve 2, 4 or 8 successive bytes for each integer. In general, using p bytes ($p = 1, 2, 4, 8$) we can represent integers in the range

Unsigned integers:	0	$\cdots$	$2^{8p} - 1$
Signed integers:	-2^{8p-1}	$\cdots$	$2^{8p-1} - 1$

Homework 2 *Compute the lower and upper bounds for signed and unsigned integers representable with $p = 2$ and with $p = 4$ bytes.*

Homework 3 (Integer overflow) *Write a program in which you define two integer variables m and i. Initialize m, read i, and print out the sum $m + i$.*

<u>Fortran version:</u>

```
program test_integer_overflow
  integer m,i
  m = 2147483645
```

```
  write(6,*) 'Please input i:'
  read*, i
  write(6,*) 'm+i=',m+i
end program test_integer_overflow
```

Matlab version:

```
function test_integer_overflow(i)
  m = int32(2147483645);
  fprintf('m+i = %d\n', m+i);
end
```

Run the program several times, with i = 1,2,3,4,5.

1. *Do you obtain correct results? What you see here is an example of* integer overflow. *The result of the summation is larger than the maximum representable integer.*

2. *What exactly happens at integer overflow? In which sense are the results inaccurate?*

3. *How many bytes does Fortran use to represent integers ?*

4. *Modify the program to print -m-i and repeat the procedure. What is the minimum (negative) integer representable?*

Note. *Except for the overflow situation, the result of an integer addition or multiplication is always exact (i.e. the numerical result is exactly the mathematical result).*

3 Floating-Point Numbers

3.1 Scientific representation of real numbers

- For most applications in science and engineering integer numbers are not sufficient; we need to work with real numbers.

- Real numbers like π have an infinite number of decimal digits; there is no hope to store them exactly.

- On a computer, floating point convention is used to represent (approximations of) the real numbers. The design of computer systems requires in-depth knowledge about floating point . Modern processors have special floating point instructions, compilers must generate such floating point instructions, and the operating system must handle the exception conditions generated by these floating point instructions.

- We now illustrate the floating point representation in base 10. Any decimal number x can be *uniquely* written as:

$x = s \cdot m \cdot 10^e$		
s	+1 or -1	sign
m	$1 \le m < 10$	mantissa
e	integer	exponent

For example

$$107.625 = +1 \cdot 1.07625 \cdot 10^2 \ .$$

- If we did not impose the *normalization condition* $1 \le m < 10$ we could have represented the number in various different ways, for example

$$(+1) \cdot 0.107625 \cdot 10^3 \ \text{or} \ \ (+1) \cdot 0.00107625 \cdot 10^5 \ .$$

- When the condition $1 \le m < 10$ is satisfied, we say that the mantissa *is normalized*. Normalization guarantees that

  1. the floating point representation is unique,
  2. since $m < 10$ there is exactly one digit before the decimal point, and
  3. since $m \ge 1$ the first digit in the mantissa is nonzero. Thus, none of the available digits is wasted by storing leading zeros.

3.2 Floating-point: representation of real numbers on finite real estate

- The amount of space used to represent each number in a machine is finite.

- Suppose our storage space is limited to 6 decimal digits per floating point number. We allocate 1 decimal digit for the sign, 3 decimal digits for the mantissa and 2 decimal digits for the exponent. If the mantissa is longer we will chop it to the most significant 3 digits (another possibility is rounding, which we will talk about shortly).

- Our example number can be then represented as

- *A floating point number is represented as (sign, mantissa, exponent) with a limited number of digits for the mantissa and the exponent.*

- The parameters of the floating point system are $\beta = 10$ (the basis), $d_m = 3$ (the number of digits in the mantissa) and $d_e = 2$ (the number of digits for the exponent).

3.3 Roundoff (approximation) errors

- Most real numbers cannot be exactly represented as floating point numbers and an approximation is needed.

- For example, numbers with an infinite representation, like $\pi = 3.141592\ldots$, will need to be "approximated" by a finite-length floating point number. In our floating point system, π will be represented as:

$$\pi = 3.141592\ldots \quad \approx \quad f\ell(\pi) = \boxed{+}\boxed{314}\boxed{00}$$

  Note that the finite representation in binary is different than finite representation in decimal; for example, $(0.1)_{10}$ has an infinite binary representation.

- In general, the floating point representation $f\ell(x)$ is just an approximation of the real number x.

- The *roundoff error* is the *relative error* is the difference between the two numbers, divided by the real number

$$\delta = \frac{f\ell(x) - x}{x} \qquad \Longleftrightarrow \qquad f\ell(x) = x \cdot (1+\delta) \ .$$

- For example, if $x = 107.625$, and $f\ell(x) = 1.07 \times 10^2$ is its representation in our floating point system, then the relative error is:

$$\delta = \frac{1.07 \times 10^2 - 107.625}{107.625} \approx -5.8 \times 10^{-3}$$

3.4 Details of the floating point representation

- With normalized mantissas, the three digits $\boxed{m_1 m_2 m_3}$ always read $m_1.m_2 m_3$, i.e. the decimal point has fixed position inside the mantissa. For the original number, the decimal point can be floated to any position in the bit-string we like by changing the exponent.

- We see now the origin of the term *floating point*: the decimal point can be floated to any position in the bit-string we like by changing the exponent.

- With 3 decimal digits, our mantissas range between $1.00, \ldots, 9.99$. For exponents, two digits will provide the range $00, \ldots, 99$.

- Consider the number 0.000123. When we represent it in our floating point system, we lose all the significant information:

$$\underbrace{\boxed{+1}}_{s} \quad \underbrace{\boxed{000}}_{m} \quad \underbrace{\boxed{00}}_{e}$$

  In order to overcome this problem, we need to allow for negative exponents also.

- We will use a *biased representation* for exponents: if the bits $e_1 e_2$ are stored in the exponent field, the actual exponent is $e_1 e_2 - 49$ (49 is called *the exponent bias*). This implies that, instead of going from 00 to 99, our exponents will actually range from -49 to $+50$. The number
$$0.000123 = +1 \cdot 1.23 \cdot 10^{-4}$$
is then represented, with the biased exponent convention, as
$$\underbrace{\boxed{+1}}_{s} \quad \underbrace{\boxed{123}}_{m} \quad \underbrace{\boxed{45}}_{e}$$

- Note that for the exponent bias we have chosen 49 and not 50. The reason for this is self-consistency: the inverse of the smallest normal number does not overflow
$$x_{\rm min} = 1.00 \times 10^{-49}\,, \qquad \frac{1}{x_{\rm min}} = 10^{+49} < 9.99 \times 10^{50} = x_{\rm max}\,.$$
(with a bias of 50 we would have had $1/x_{\rm min} = 10^{50} > 9.99 \times 10^{+49} = x_{\rm max}$).

- What is the maximum number allowed by our toy floating point system? If $m = 9.99$ and $e = +99$, we obtain

$$x = 9.99 \cdot 10^{50} \ .$$

  If $m = 000$ and $e = 00$ we obtain a representation of ZERO. Depending on S, it can be $+0$ or -0. Both numbers are valid, and we will consider them equal.

- What is the minimum positive number that can be represented in our toy floating point system? The smallest mantissa value that satisfies the normalization requirement is $m = 1.00$; together with $e = 00$ this gives the number 10^{-49}.

- If we drop the normalization requirement, we can represent smaller numbers also. For example, $m = 0.10$ and $e = 00$ give 10^{-50}, while $m = 0.01$ and $e = 00$ give 10^{-51}.

  The floating point numbers with exponent equal to ZERO and the first digit in the mantissa also equal to ZERO are called subnormal numbers.

Allowing subnormal numbers improves the resolution of the floating point system near 0. Non-normalized mantissas will be permitted only when $e = 00$, to represent ZERO or subnormal numbers, or when $e = 99$ to represent special numbers.

- Example (D. Goldberg, p. 185, adapted): Suppose we work with our toy floating point system and do not allow for subnormal numbers. Consider the fragment of code

$$IF\ (x \neq y)\ THEN\ z = 1.0/(x - y)$$

designed to "guard" against division by 0. Let $x = 1.02 \times 10^{-49}$ and $y = 1.01 \times 10^{-49}$. Clearly $x \neq y$ but, (since we do not use subnormal numbers) $x \ominus y = 0$. In spite of all the trouble we are dividing by 0! If we allow subnormal numbers, $x \ominus y = 0.01 \times 10^{-49}$ and the code behaves correctly.

3.5 Binary floating point numbers

- Similar to the decimal case, any binary number x can be represented as:

$x = s \cdot m \cdot 2^e$		
s	+1 or -1	sign
m	$1 \leq m < 2$	mantissa
e	integer	exponent

- For example,

$$1101011.101 = +1 \cdot 1.101011101 \cdot 2^6 \ . \tag{1}$$

With 6 binary digits available for the mantissa and 4 binary digits available for the exponent, the floating point representation is

$$\underbrace{\boxed{+1}}_{s}\ \underbrace{\boxed{110101}}_{m}\ \underbrace{\boxed{0110}}_{e} \tag{2}$$

- When we use normalized mantissas, the first digit is always nonzero. With binary floating point representation, a nonzero digit is (of course)

1, hence the first digit in the normalized binary mantissa is always 1.

$$1 \le x < 2 \quad \rightarrow \quad (x)_2 = \mathbf{1}.m_1 m_2 m_3 \ldots$$

As a consequence, it is not necessary to store it; we can store the mantissa starting with the second digit, and store an extra, least significant bit, in the space we saved. This is called *the hidden bit technique*.

- For our binary example (2) the leftmost bit (equal to 1, of course, showed in bold) is redundant. If we do not store it any longer, we obtain the hidden bit representation:

$$\underbrace{\boxed{+1}}_{s} \; \underbrace{\boxed{10101\mathbf{1}}}_{m} \; \underbrace{\boxed{0110}}_{e} \tag{3}$$

- Hidden bit allows to pack more information in the same space: the rightmost bit of the mantissa holds now the 7th bit of the number (1) (equal to 1, showed in bold). This 7th bit was simply omitted in the standard form (2).

4 The IEEE standard

- The IEEE standard regulates the representation of binary floating point numbers in a computer, how to perform consistently arithmetic operations and how to handle exceptions, etc. Developed in 1980's, is now followed by virtually all microprocessor manufacturers.

- Supporting IEEE standard greatly enhances programs portability. When a piece of code is moved from one IEEE-standard-supporting machine to another IEEE-standard-supporting machine, the results of the basic arithmetic operations (+,-,*,/) will be identical.

4.1 Floating Point Types

The standard defines the following floating point types:

- **Single Precision.** (4 consecutive bytes/ number).

  $$\boxed{\pm \mid e_1 e_2 e_3 \cdots e_8 \mid m_1 m_2 m_3 \cdots m_{23}}$$

  Useful for most short calculations.

- **Double Precision.** (8 consecutive bytes/number)

  $$\boxed{\pm \mid e_1 e_2 e_3 \cdots e_{11} \mid m_1 m_2 m_3 \cdots m_{52}}$$

  Most often used with scientific and engineering numerical computations.

- **Extended Precision.** (10 consecutive bytes/number).

  $$\boxed{\pm | e_1 e_2 e_3 \cdots e_{15} \mid m_1 m_2 m_3 \cdots m_{64}}$$

Useful for temporary storage of intermediate results in long calculations. (e.g. compute a long inner product in extended precision then convert the result back to double)

- There is a single-extended format also. The standard suggests that implementations should support the extended format corresponding to the widest basic format supported (since all processors today allow for double precision, the double-extended format is the only one we discuss here).

- Extended precision enables libraries to efficiently compute quantities within 0.5 `ulp`. For example, the result of `x*y` is correct within 0.5 `ulp`, and so is the result of `log(x)`. Clearly, computing the logarithm is a more involved operation than multiplication; the log library function performs all the intermediate computations in extended precision, then rounds the result to single or double precision, thus avoiding the corruption of more digits and achieving a 0.5 `ulp` accuracy. From the user point of view this is transparent, the log function returns a result correct within 0.5 `ulp`, the same accuracy as simple multiplication has.

4.2 Detailed IEEE representation

- Single precision standard (double precision is similar)

$$\boxed{\pm | e_1 e_2 e_3 \cdots e_8 | m_1 m_2 m_3 \cdots m_{23}}$$

- **Signum.** "±" bit = 0 (positive) or 1 (negative).
- **Exponent.** Biased representation, with an exponent bias of $(127)_{10}$.
- **Mantissa.** Hidden bit technique.
- Detailed format is shown in Table 1.

4.3 Number range

The range of numbers represented in different IEEE formats is summarized in Table 2.

$e_1 e_2 e_3 \cdots e_8$	Represented value
$(00000000)_2 = (0)_{10}$	$\pm(0.m_1 \ldots m_{23})_2 \times 2^{-126}$ (ZERO or subnormal)
$(00000001)_2 = (1)_{10}$	$\pm(1.m_1 \ldots m_{23})_2 \times 2^{-126}$
$\cdots$	$\cdots$
$(01111111)_2 = (127)_{10}$	$\pm(1.m_1 \ldots m_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.m_1 \ldots m_{23})_2 \times 2^{1}$
$\cdots$	$\cdots$
$(11111110)_2 = (254)_{10}$	$\pm(1.m_1 \ldots m_{23})_2 \times 2^{+127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $m_1 \ldots m_{23} = 0$ NaN otherwise

Table 1: Details on single precision IEEE floating point format. Note that $-e_{\min} < e_{\max}$, which implies that $1/x_{\min}$ does not overflow.

IEEE Format	$E_{\min}$	$E_{\max}$
Single Precision	-126	+127
Double Precision	-1,022	+1,023
Extended Precision	-16,383	+16,383

Table 2: IEEE floating point number exponent ranges

4.4 Precision

- **The precision of the floating point system (the "machine precision") is the smallest number ϵ for which**

$$1 \oplus \epsilon > 1 \qquad (\text{meaning } 1 \oplus \epsilon \neq 1).$$

- To understand the *precision of the floating point system*, let us go back to our toy floating point representation (2 decimal digits for the exponent and 3 for the mantissa).

  We want to add two numbers, e.g.

$$1 = 1.00 \times 10^0 \text{ and } 0.01 = 1.00 \times 10^{-2} \ .$$

  In order to perform the addition, we bring the smaller number to the same exponent as the larger number *by shifting right the mantissa*. For our example,

$$1.00 \times 10^{-2} = 0.01 \times 10^0 \ .$$

Next, we add the mantissas and normalize the result if necessary. In our case

$$1.00 \times 10^0 + 0.01 \times 10^0 = 1.01 \times 10^0 \ .$$

Suppose now we want to add

$$1 = 1.00 \times 10^0 \text{ and } 0.001 = 1.00 \times 10^{-3} \ .$$

For bringing them to the same exponent, we need to shift right the mantissa 3 positions, and, due to our limited space (3 digits) we lose all the significant information. Thus

$$1.00 \times 10^0 + 0.00[1] \times 10^0 = 1.00 \times 10^0 \ .$$

We can see now that this is a limitation of the floating point system due to the storage of only a finite number of digits.

- For our toy floating point system, it is clear from the previous discussion that $\epsilon = 0.01$.

- If the relative error in a computation is $p\epsilon$, then the number of corrupted decimal digits is $\log_{10} p$.

- In binary IEEE arithmetic:
  - The first single precision number larger than 1 is $1 + 2^{-23}$;
  - The first double precision number larger than 1 is $1 + 2^{-52}$;
  - For extended precision there is no hidden bit, so the first such number is $1 + 2^{-63}$.

  You should be able to justify these yourselves. The machine precision for different floating point formats is described in Table 4.4. Note that Matlab offers a predefined variable `eps` that equals the machine precision of the double precision format.

- If the relative error in a computation is $p\epsilon$, then the number of corrupted binary digits is $\log_2 p$.

Remark 1 *We can now answer the following question. Signed integers are represented in two's complement. Signed mantissas are represented using the sign-magnitude convention. For signed exponents the standard uses a biased representation. Why not represent the exponents in two's complement, as we do for the signed integers? When we compare two floating*

IEEE Format	Machine precision (ϵ)	No. Decimal Digits
Single Precision	$2^{-23} \approx 1.2 \times 10^{-7}$	7
Double Precision	$2^{-52} \approx 1.1 \times 10^{-16}$	16
Extended Precision	$2^{-63} \approx 1.1 \times 10^{-19}$	19

Table 3: Precision of different IEEE representations

point numbers (both positive, for now) the exponents are looked at first; only if they are equal we proceed with the mantissas. The biased exponent is a much more convenient representation for the purpose of comparison. We compare two signed integers in greater than/less than/ equal to expressions; such expressions appear infrequently enough in a program, so we can live with the two's complement formulation, which has other benefits. On the other hand, any time we perform a floating point addition/subtraction we need to compare the exponents and align the operands. Exponent comparisons are therefore quite frequent, and being able to do them efficiently is very important. This is the argument for preferring the biased exponent representation.

Homework 4 *Consider the real number* $(0.1)_{10}$. *Write its single preci-*

sion, floating point representation. Does the hidden bit technique result in a more accurate representation?

Homework 5 *What is the gap between 1024 and the first IEEE single precision number larger than 1024?*

Homework 6 *Let* $x = m \times 2^e$ *be a normalized single precision number, with* $1 \leq m < 2$. *Show that the gap between* x *and the next largest single precision number is*

$$\epsilon \times 2^e \ .$$

Homework 7 *The following program adds* $1 + 2^{-p}$, *then subtracts* 1. *If* $2^{-p} < \epsilon$ *the final result will be zero. By providing different values for the exponent, you can find the machine precision for single and double precision. Note the declaration for the simple precision variables ("real") and the declaration for double precision variables ("double precision"). The command* $2.0 * *p$ *calculates* 2^p *(** is the power operator). Also note the form of the constants in single precision (2.e0) vs. double precision (2.d0).*

<u>**Fortran version:**</u>

```
program test_precision
```

```
   real :: a
   double precision :: b
   integer :: p
   print*, 'Please provide exponent'
   read*, p
   a = 1.e0 + 2.e0**(-p)
   print*, a-1.e0
   b = 1.d0 + 2.d0**(-p)
   print*, b-1.d0
end
```

Matlab version:

```
function test_precision(p)
% p = number of matissa bits
   a = single(1.0 + 2.0^(-p));
   fprintf('Single precision: a - 1 = %e\n',a-1);
   b = double(1.0 + 2.0^(-p));
   fprintf('Double precision: b - 1 = %e\n',b-1);
   fprintf('Matlab''s predefined epsilon machine: eps = %e\n',eps);
end
```

Run the program for values different of p ranging from 20 to 60. Find experimentally the values of ϵ for single and for double precision.

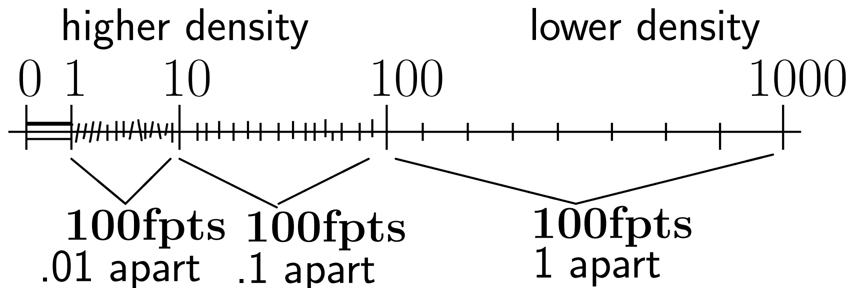
5 The Set of Floating Point Numbers

- The set of all floating point numbers consists of:

$$\text{floating point} = \{\ \pm 0\ ,\ \text{all normal}\ ,\ \text{all subnormal}\ ,\ \pm\infty\ \}.$$

- Because of the limited number of digits, the floating point numbers are *a finite set*. For example, in our toy floating point system, we have approximately $2 \cdot 10^5$ floating point numbers altogether.

- The floating point numbers are not uniformly spread between min and max values; they have a high density near zero, but get sparser as we move away from zero.



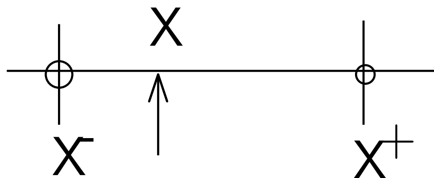
For example, in our floating point system, there are 90 points between 1 and 10 (hence, the gap between 2 successive numbers is 0.01). Between 10 and 100 there are again 90 floating point numbers, now with a gap of 0.1. The interval 100 to 1000 is "covered" by another 90 floating point values, the difference between 2 successive ones being 1.0.

- In general, if $m \times 10^e$ is a normalized floating point number, with mantissa $1.00 \leq m < 9.98$, the very next floating point number representable is $(m + \epsilon) \times 10^e$ (please give a moment's thought about why this is so). In consequence, the gap between $m \times 10^e$ and the next floating point number is $\epsilon \times 10^e$. The larger the floating point numbers, the larger the gap between them will be (the machine precision ϵ is a fixed number, while the exponent e increases with the number).

- In binary format, similar expressions hold. Namely, the gap between $m \times 2^e$ and its successor is $\epsilon \times 2^e$.

6 Rounding

It is often the case that we have a real number X that is not exactly a floating point number: X falls between two consecutive floating point numbers X^- and X^+.



In order to represent a real number X in the computer we need to approximate it by a floating point number.

6.1 Rounding up or down

- If we choose X^- we say that we *rounded* X *down*; if we choose X^+ we say that we *rounded* X *up*. We can choose a different floating point number also, but this makes little sense, as the approximation error will be larger than with $X^\pm$.

For example, $\pi = 3.141592\ldots$ is in between $\pi^- = 3.14$ and $\pi^+ = 3.15$. π^- and π^+ are successive floating point numbers in our toy system.

- We will denote $f\ell(X)$ the floating point number that approximates X. Then

$$f\ell(X) = \begin{cases} X^- \, , & \text{if rounding down,} \\ X^+ \, , & \text{if rounding up.} \end{cases}$$

- When rounding up or down we make a certain representation error; we call it **the roundoff (rounding) error**.

  The relative roundoff error, δ, is defined as

$$\delta = \frac{f\ell(X) - X}{X} \, .$$

  This does not work for $X = 0$, so we will prefer the equivalent formulation

$$f\ell(X) = X \cdot (1 + \delta) \, .$$

- What is the largest error that we can make when rounding (up or down)? The two floating point candidates can be represented as

$X^- = m \times 2^e$ and $X^+ = (m + \epsilon) \times 2^e$ (this is correct since they are successive floating point numbers). For now suppose both numbers are positive (if negative, a similar reasoning applies). Since

$$|f\ell(X) - X| \le |X^+ - X^-|, \text{ and } X \ge X^- ,$$

we have

$$|\delta| \le \frac{|X^+ - X^-|}{X^-} = \frac{\epsilon \times 2^e}{m \times 2^e} \le \epsilon.$$

Homework 8 *Find an example of X such that, in our toy floating point system, rounding down produces a roundoff error $\delta = \epsilon$. This shows that, in the worst case, the upper bound ϵ can actually be attained.*

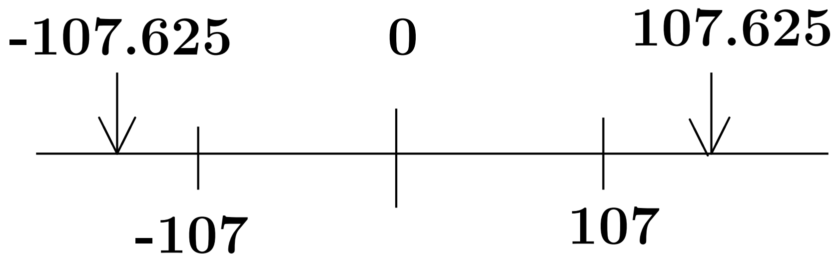
Now, we need to choose which one of X^+, X^- 'better" approximates X. There are two possible approaches.

6.2 Rounding to zero ("chopping")

- Suppose $X = 107.625$. We can represent it as $\boxed{+1}\boxed{107}\boxed{+2}$ by simply discarding ("chopping") the digits which do not fit the mantissa

format (here the remaining digits are 625). We see that the floating point representation is precisely X^-, and we have $0 \le X^- < X$. Now, if X was negative, $X = -107.625$, the chopped representation would be | -1 | 107 | +2 |, but now this is X^+. Note that in this situation $X < X^+ \le 0$.

- In consequence, with chopping, we choose X^- if $X > 0$ and X^+ is $X < 0$. In both situations the floating point number is closer to 0 than the real number X, so chopping is also called *rounding toward 0*.



- Chopping has the advantage of being very simple to implement in

hardware. The roundoff error for chopping satisfies

$$-\epsilon < \delta_{\text{chopping}} \le 0 \ .$$

For example:

$$X = 1.00999999\ldots \Rightarrow f\ell(X)_{\text{chop}} = 1.00$$

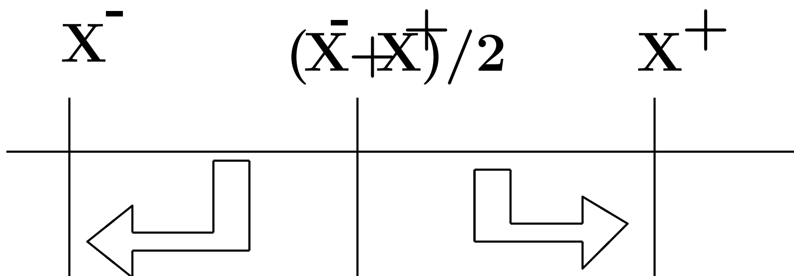
and

$$\delta = \frac{f\ell(X) - X}{X} = \frac{-0.0099...}{1.00999...} = -0.0099 \approx -0.01 = \epsilon \ .$$

6.3 Rounding to nearest

- This approximation mode is used by most processors, and is called, in short *"rounding"*.
- The idea is to choose the floating point number (X^- or X^+) which offers the best approximation of X:

$$f\ell(X) = \begin{cases} X^- \ , & \text{if } X^- \le X < \frac{X^+ + X^-}{2}, \\ X^+ \ , & \text{if } \frac{X^+ + X^-}{2} < X \le X^+. \end{cases}$$



- The roundoff for the "round to nearest" approximation mode satisfies

$$-\frac{\epsilon}{2} \leq \delta_{\text{rounding}} \leq \frac{\epsilon}{2}\,.$$

  The worst possible error is here half (in absolute magnitude) the worst-case error of chopping.

- In addition, the errors in the "round to nearest" approximation have both positive and negative signs. Thus, when performing long computations, it is likely that positive and negative errors will cancel each other out, giving a better numerical performance with "rounding" than with "chopping".

- There is a fine point to be made regarding "round to nearest" approximation. What happens if there is a tie, i.e. if X is precisely $(X^+ + X^-)/2$? For example, with 6 digits mantissa, the binary number $X = 1.0000001$ can be rounded to $X^- = 1.000000$ or to $X^+ = 1.000001$. In this case, the IEEE standard requires to choose the approximation with an even last bit; that is, here choose X^-. This ensures that, when we have ties, half the roundings will be done up and half down.

  The idea of rounding to even can be applied to decimal numbers also (and, in general, to any basis). To see why rounding to even works better, consider the following example. Let $x = 5 \times 10^{-2}$ and compute $((((1 \oplus x) \ominus x) \oplus x) \ominus x)$ with correct rounding. All operations produce exact intermediate results with the fourth digit equal to 5; when rounding this exact result, we can go to the nearest even number, or we can round up, as is customary in mathematics. Rounding to nearest even produces the correct result (1.00), while rounding up produces 1.02.

- An alternative to rounding is *interval arithmetic*. The output of an operation is an interval that contains the correct result. For example

$x \oplus y \in [\underline{z}, \bar{z}]$, where the limits of the interval are obtain by rounding down and up respectively. The final result with interval arithmetic is an interval that contains the true solution; if the interval is too large to be meaningful we should repeat the calculations with a higher precision.

Homework 9 *In IEEE single precision, what are the rounded values for* $4 + 2^{-20}$, $8 + 2^{-20}$, $16 + 2^{-20}$, $32 + 2^{-20}$, $64 + 2^{-20}$. *(Here and from now "rounded" means "rounded to nearest".)*

6.4 Summary of rounding modes

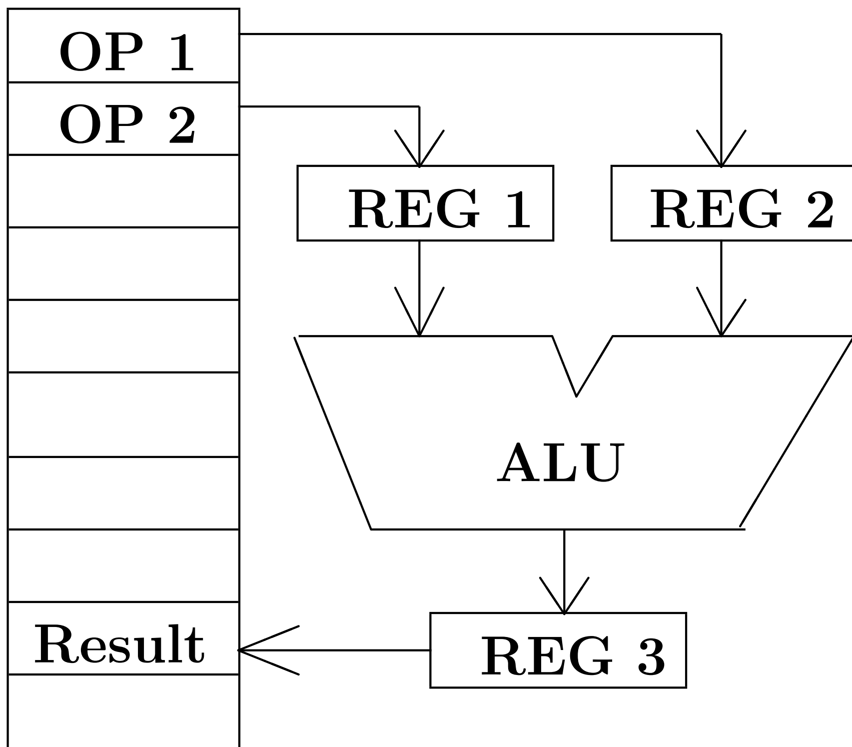
In conclusion, real numbers are approximated and represented in the floating point format. The IEEE standard recognizes four approximation modes:

1. Round Up;
2. Round Down;
3. Round Toward Zero;
4. Round to Nearest (Even).

Virtually all processors implement the ("round to nearest") approximation. From now on, we will call it, by default, *"rounding"*. Computer numbers are therefore accurate only within a factor of $(1 \pm \epsilon/2)$. In single precision, this gives 1 ± 10^{-7}, or about 7 accurate decimal places. In double precision, this gives 1 ± 10^{-16}, or about 16 accurate decimal digits.

7 Arithmetic Operations

- To perform arithmetic operations, the values of the operands are loaded into registers; the Arithmetic and Logic Unit (ALU) performs the operation, and puts the result in a third register; the value is then stored back in memory.


OP 1
OP 2
REG 1
REG 2
ALU
REG 3
Result

- The two operands are obviously floating point numbers. The result of the operation stored in memory must also be a floating point number.

- Is there any problem here? Yes! Even if the operands are floating point numbers, the result of an arithmetic operation may not be a floating point number.

  To understand this, let us add two floating point numbers, a= | 9.72 | 01 | (97.2) and b= | 6.43 | 00 | (6.43), using our toy floating point system. To perform the summation we need to align the numbers by shifting the smaller one (6.43) to the right.

$$
\begin{array}{rcccc}
9. & 7 & 2 & & 01 \\
0. & 6 & 4 & 3 & 01 \\
\hline
10. & 3 & 6 & 3 & 01
\end{array}
$$

  The result (103.63) is not a floating number. We can round the result to obtain | 1.04 | 02 | (104).

- From this example we draw a first useful conclusion: the result of any arithmetic operation is, in general, corrupted by roundoff errors. Thus, the arithmetic result is different from the mathematical result.

- If a, b are floating point numbers, and $a + b$ is the result of mathematical addition, we will denote by $a \oplus b$ the computed addition.

- The fact that $a \oplus b \neq a+b$ has surprising consequences. Let c= $\boxed{4.99}\boxed{-1}$ (0.499). Then

$$\underbrace{(a \oplus b)}_{104} \oplus \underbrace{c}_{0.499} = f\ell(104.499) = \boxed{1.04}\boxed{02}(104),$$

while

$$\underbrace{a}_{97.2} \oplus \underbrace{(b \oplus c)}_{f\ell(6.529)} = \boxed{1.05}\boxed{02}(105)$$

(you can readily verify this). Unlike mathematical addition, computed addition is *not associative*!

Homework 10 *Show that computed addition is commutative, i.e.* $a \oplus b = b \oplus a$.

7.1 IEEE Arithmetic

- The IEEE standard specifies that the result of an arithmetic operation (+,-,*,/) must be computed exactly and then rounded to nearest. In other words,

$$
\begin{aligned}
a \oplus b &= f\ell(a+b) \\
a \ominus b &= f\ell(a-b) \\
a \otimes b &= f\ell(a \times b) \\
a \oslash b &= f\ell(a/b) \ .
\end{aligned}
$$

  The same requirement holds for square root, remainder, and conversions between integer and floating point formats: compute the result exactly, then round.

- This IEEE convention completely specifies the result of arithmetic operations; operations performed in this manner are called *exactly*, or *correctly rounded*. It is easy to move a program from one machine that supports IEEE arithmetic to another. Since the results of arithmetic operations are completely specified, all the intermediate results

should coincide to the last bit (if this does not happen, we should look for software errors!).

- Note that it would be nice to have the results of transcendental functions like `exp(x)` computed exactly, then rounded to the desired precision; this is however impractical, and the standard does NOT require correctly rounded results in this situation.

- Performing only correctly rounded operations seems like a natural requirement, but it is often difficult to implement it in hardware. The reason is that if we are to find first the exact result we may need additional resources. Sometimes it is not at all possible to have the exact result in hand - for example, if the exact result is a periodic number (in our toy system, $2.0/3.0 = 0.666...$).

8 Pitfalls with Floating Point Arithmetic

8.1 Binary versus decimal

Homework 11 *Run the following code fragment.*
Fortran version:

```
program test_conversion_1
  real x
  data x /1.0E-4/
  print*, x
end program test
```

Matlab version:

```
x = single(1.e-4);
format long
disp(x)
```

We expect the answer to be $1.0E-4$, but in fact the program prints $9.9999997E-05$. Note that we did nothing but store and print! The "anomaly" comes from the fact that 0.0001 is converted (inexactly) to binary, then the stored binary value is converted back to decimal for printing.

8.2 Floating point comparisons

Because of the inexact results it is best to avoid strict equality when comparing floating point numbers.

Homework 12 (Fp_comparison) *Consider the following code.*
Fortran version:

```
x = 1.0e-4
if ( 1.0e+8 * x**2 == 1.0 ) then
   print*, 'Correct'
else
   print*, 'Bizarre'
end if
```

Matlab version:

```
x = single(1.e-4);
if ( 1.0e+8 * x^2 == 1.0 )
   disp('Correct')
else
   disp('Bizarre')
end
```

should print ``Correct'', *but does not, since the left expression is corrupted by roundoff.*

The right way to do floating point comparisons is to define the epsilon machine, eps, *and check that the magnitude of the difference is less than half epsilon times the sum of the operands.*

Fortran version:

```
epsilon = 1.192093e-07
w = 1.0e+8 * x**2
if ( abs(w-1.0) <= 0.5*epsilon*( abs(w)+abs(1.0) ) ) then
   print*, 'Correct'
else
   print*, 'Bizarre'
end if
```

Matlab version:

```
x = single(1.0e-4);
epsilon = 1.192093e-07;
w = 1.0e+8 * x^2;
if ( abs(w-1.0) <= 0.5*epsilon*( abs(w)+abs(1.0) ) )
   disp('Correct')
```

```
else
   disp('Bizarre')
end
```

This time we allow small roundoff's to appear, and the program takes the right branch.

8.3 Funny conversions

Sometimes the inexactness in floating point is uncovered by real to integer conversion, which by Fortran default is done using truncation. For example the code

```
program test_conversion_1
  real x
  integer i
  data x /0.0001/
  i = 10000*x
  print *, i
end program test
```

produces a stunning result: the value of i is 0, not 1!

Matlab shields us from these effects:

```
x = single(1.e-4);
t = single(10000);
i = int32(t*x);
disp(i)
j = floor(t*x);
disp(j)
k = ceil(t*x);
disp(k)
```

Another problem appears when a single precision number is converted to double precision. This does not increase the accuracy of the number.

Homework 13 (test_conversion_2) *Run the following code.*

Fortran version:

```
program test_conversion_2
  real x
  double precision y
  data x /1.234567/
```

```
  y = x
  print *, 'X =',x,'  Y =',y
end program test
```

Matlab version:

```
  x = single(1.234567);
  y = double(x);
  format long
  disp(x);
  disp(y);
```

The code produces the output

```
 X=   1.234567  Y=   1.23456704616547
```

The explanation is that, when converting single to double precision, register entries are padded with zeros in the binary representation. The double precision number is printed with 15 positions and the inexactity shows up. (if we use a formatted print for `x` with 15 decimal places we obtain the same result). In conclusion, we should only print the number of digits that are significant to the problem.

8.4 Memory versus register operands

The code

```
data a /3.0/, b /10.0/
data x /3.0/, y /10.0/
z = (y/x)-(b/a)
call ratio(x,y,a1)
call ratio(a,b,a2)
call sub(a2,a1,c)
print*, z-c
```

may produce a nonzero result. This is so because `z` is computed with register operands (and floating point registers for Intel are in extended precision, 80 bits) while for `c` the operands `a` and `b` are stored in the memory.

8.5 Cancellation ("Loss-of Significance") Errors

When subtracting numbers that are nearly equal, the most significant digits in the operands match and cancel each other. This is no problem if the

operands are exact, but in real life the operands are corrupted by errors. In this case the cancellations may prove catastrophic.

For example, we want to solve the quadratic equation

$$a\,x^2 + b\,x + c = 0\;,$$

where all the coefficients are floating point numbers

$$a = 1.00 \times 10^{-3},\; b = 1.00 \times 10^{0},\; c = 9.99 \times 10^{-1}\;,$$

using our toy decimal floating point system and the quadratic formula

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\;.$$

The true solutions are $r_1 = -999$, $r_2 = -1$. In our floating point system $b^2 = 1.00$, $4ac = 3.99 \times 10^{-3}$, and $b^2 - 4ac = 1.00$. It is here where the cancellation occurs! Then $r_1 = (-1-1)/(2 \times 10^{-3}) = -1000$ and $r_2 = (-1+1)/(2 \times 10^{-3}) = 0$. If the error in r_1 is acceptable, the error in r_2 is 100%!

To overcome this, we might use the pair of mathematically equivalent formulas

$$r_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}\;.$$

With this formula, $r_2 = (2c)/(-2) = -9.99 \times 10^{-1}$, a much better approximation.

Homework 14 (quadratic_equation) *Run the following Matlab code:*

```
a = single(1.0);
b = single(-1e8);
c = single(9.999999e7);
four = single(4.0);
two  = single(2.0);
d  = sqrt(b^2 - four*a*c);
r1 = (-b+d)/(two*a);
r2 = (-b-d)/(two*a);
format long
disp(r1);
disp(r2);
```

The computed roots in single precision are 100000000 and 0. The computed roots in double precision are $9.999999900000009e{+}07$ and 0.999999910593033.

8.6 Insignificant Digits

Homework 15 (insignificant_digits) *Run the Fortran code*

```
program test_insignificant_digits
  real :: x=100000.0, y=100000.1, z
  z = y-x
  print*, 'Z=',z
end program test_insignificant_digits
```

or its Matlab version

```
x = single(123456.7);
y = single(123456.8);
z = y-x;
fprintf('z = %f\n', z);
```

We would expect the output

$$Z = 0.1000000$$

but in fact the program prints (on Alpha ...)

$$Z = 0.1015625$$

Since single precision handles about 7 decimal digits, and the subtraction $z = y - x$ cancels the most significant 6, the result contains only one significant digit. The appended garbage `15625` are insignificant digits, coming from the inexact binary representation of `x` and `y`. Beware of convincing-looking results!

8.7 Order matters

Mathematically equivalent expressions may give different values in floating point, depending on the order of evaluation.

Homework 16 (operation_order) *Run the Fortran code*

```
program test_operation_order
  real :: x=12345.6, y=45678.9, z=98765432.1
  real :: w1, w2
  w1 = x*y/z
  w2 = 1/z; w2 = x*w2; w2 = y*w2
  print*, w1-w2
end program test_operation_order
```

or its Matlab version

```
x = single(12345.6);
y = single(45678.9);
z = single(98765432.1);
one = single(1.0);
w1 = x*y/z;
w2 = y*(x*(one/z));
format long
disp(w1-w2);
```

Mathematically, the difference between `w1` *and* `w2` *should be zero, but ... it is about* $-4.e-7$*.*

9 Integer Multiplication

- As another example, consider the multiplication of two single-precision, floating point numbers.

$$(m_1 \times 2^{e_1}) \cdot (m_2 \times 2^{e_2}) = (m_1 \cdot m_2) \times 2^{e_1+e_2} \ .$$

- In general, the multiplication of two 24-bit binary numbers $(m_1 \cdot m_2)$ gives a 48-bit result. This can be easily achieved if we do the multiplication in double precision (where the mantissa has 53 available bits), then round the result back to single precision.

- However, if the two numbers to be multiplied are double-precision, the exact result needs a 106-bit long mantissa; this is more than even extended precision can provide. Usually, multiplications and divisions are performed by specialized hardware, able to handle this kind of problems.

10 Special Arithmetic Operations

10.1 Signed zeros

- Recall that the binary representation 0 has all mantissa and exponent bits zero. Depending on the sign bit, we may have $+0$ or -0. Both are legal, and they are *distinct*; however, if $x = +0$ and $y = -0$ then the comparison (x.`EQ`.y) returns `.TRUE.` for consistency.

- The main reason for allowing signed zeros is to maintain consistency with the two types of infinity, $+\infty$ and $-\infty$. In IEEE arithmetic, $1/(+0) = +\infty$ and $1/(-0) = -\infty$. If we had a single, unsigned 0, with $1/0 = +\infty$, then $1/(1/-\infty) = 1/0 = +\infty$, and not $-\infty$ as expected.

- There are other good arguments in favor of signed zeros. For example, consider the function $\tan(\pi/2 - x)$, discontinuous at $x = 0$; we can consistently define the result to be $\mp\infty$ based on the signum of $x = \pm 0$.

- Signed zeros have disadvantages also; for example, with $x = +0$ and

$y = -0$ we have that $x = y$ but $1/x \neq 1/y$!

10.2 Operations with ∞

The following operations with infinity are possible:

$a/(+\infty)$	$=$	$\begin{cases} 0, & a \text{ finite} \\ \text{NaN}, & a = \infty \end{cases}$
$a * (+\infty)$	$=$	$\begin{cases} +\infty\ , & a > 0\ , \\ -\infty\ , & a < 0\ , \\ NaN, & a = 0\ . \end{cases}$
$+\infty + a$	$=$	$\begin{cases} \infty\ , & a \text{ finite}\ , \\ \text{NaN}, & a = -\infty\ . \end{cases}$

10.3 Operations with `NaN`

- Any operation involving NaN as (one of) the operand(s) produces NaN.
- In addition, the following operations "produce" NaN: $\infty + (-\infty)$, $0 * \infty$, $0/0$, ∞/∞, $\sqrt{-|x|}$, x *modulo* 0, ∞ *modulo* x.

10.4 Comparisons

The IEEE results to comparisons are summarized below:

$(a < b)$.OR.$(a = b)$.OR.$(a > b)$	True, if a, b floating point numbers False, if one of them NaN
$+0 = -0$	True
$+\infty = -\infty$	False

11 Arithmetic Exceptions

One of the most difficult things in programming is to treat exceptional situations. It is desirable that a program handles exceptional data in a manner consistent with the handling of normal data. The results will then provide the user with the information needed to debug the code, if an exception occurred. The extra floating point numbers allowed by the IEEE standard are meant to help handling such situations.

The IEEE standard defines 5 exception types: division by 0, overflow, underflow, invalid operation and inexact operation.

11.1 Division by 0

If a is a floating point number, then IEEE standard requires that

$$\frac{a}{0.0} = \begin{cases} +\infty\ , & \text{if } a > 0\ , \\ -\infty\ , & \text{if } a < 0\ , \\ \text{NaN}\ , & \text{if } a = 0. \end{cases}$$

If $a > 0$ or $a < 0$ the ∞ definitions make mathematical sense. Recall that $\pm\infty$ have special binary representations, with all exponent bits equal to 1 and all mantissa bits equal to 0.

If $a = 0$, then the operation is $0/0$, which makes no mathematical sense. What we obtain is therefore invalid information. The result is the **"Not a Number"**, in short **NaN**. Recall that NaN also have a special binary representation. NaN is a red flag, which tells the user that something wrong happened with the program. ∞ may or may not be the result of a bug, depending on the context.

11.2 Overflow

Occurs when the result of an arithmetic operation is finite, but larger in magnitude than the largest floating point number representable using the given precision. The standard IEEE response is to set the result to $\pm\infty$ (round to nearest) or to the largest representable floating point number (round toward 0). Some compilers will trap the overflow and abort execution with an error message.

Example (Demmel 1984, from D. Goldberg, p. 187, adapted): In our

toy floating point system let's compute

$$\frac{2 \times 10^{23} + 10^{23}\,\mathbf{i}}{2 \times 10^{25} + 10^{25}\,\mathbf{i}}$$

whose result is 1.00×10^{-2}, a "normal" floating point number. A direct use of the formula

$$\frac{a + b\,\mathbf{i}}{c + d\,\mathbf{i}} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}\,\mathbf{i}$$

returns the result equal to 0, since the denominators overflow. Using the scaled formulation

$$\xi = \frac{d}{c}\,; \qquad \frac{a + b\,\mathbf{i}}{c + d\,\mathbf{i}} = \frac{a + b\xi}{c + d\xi} + \frac{b - a\xi}{c + d\xi}\,\mathbf{i}$$

we have $\xi = 0.5$, $(a + b\xi)/(c + d\xi) = (2.5 \times 10^{23})/(2.5 \times 10^{25}) = 0.01$ and $b - a\xi = 0$.

Sometimes overflow and infinity arithmetic may lead to curious results. For example, let $x = 3.16 \times 10^{25}$ and compute

$$\frac{x^2}{(x + 1.0 \times 10^{23})^2} = 9.93 \times 10^{-1}$$

Since the denominator overflows it is set to infinity; the numerator does not overflow, therefore the result is 0!. If we compute the same quantity as

$$\left(\frac{x}{x + 1 \times 10^{23}}\right) = \left(\frac{3.16}{3.17}\right) = 0.99$$

we obtain a result closer to the mathematical value.

11.3 Underflow

Occurs when the result of an arithmetic operation is smaller than the smallest *normalized* floating point number which can be stored. In IEEE standard the result is a *subnormal* number ("gradual" underflow) or 0, if the result is small enough. Note that subnormal numbers have fewer bits of precision than normalized ones, so using them may lead to a *loss of accuracy*. For example, let

$$x = 1.99 \times 10^{-40}\ , \quad y = 1.00 \times 10^{-11}\ , \quad z = 1.00 \times 10^{+11}\ ,$$

and compute $t = (x \otimes y) \otimes z$. The mathematical result is $t = 1.99 \times 10^{-40}$. According to our roundoff error analysis, we expect the calculated t to

satisfy

$$\hat{t}_{\rm expected} = (1+\delta) t_{\rm exact}~, \quad |\delta| \approx \epsilon~,$$

where the bound on delta comes from the fact that we have two floating point multiplications, and (with exact rounding) each of them can introduce a roundoff error as large as the half the machine precision $|\delta_\otimes| \le \epsilon/2$:

$$\begin{aligned}
x \otimes y &= (1+\delta_\otimes^1)(x \times y) \\
(x \otimes y) \otimes z &= (1+\delta_\otimes^2)\,[(x \otimes y) \times z] \\
&= (1+\delta_\otimes^2)(1+\delta_\otimes^1)\,[x \times y \times z] \\
&\approx (1+\delta_\otimes^1+\delta_\otimes^2)\,[x \times y \times z] \\
&\le (1+\epsilon)\,[x \times y \times z]
\end{aligned}$$

Since in our toy system $\epsilon = 0.01$, we expect the computed result to be in the range

$$\hat{t}_{\rm expected} \in [(1-2\epsilon) t_{\rm exact},\ (1+2\epsilon) t_{\rm exact}] = [1.98 \times 10^{-40},\ 2.00 \times 10^{-40}]~.$$

However, the product $x \otimes y = 1.99 \times 10^{-51}$ underflows, and has to be represented by the subnormal number 0.01×10^{-49}; when multiplied by z

this gives $\hat{t} = 1.00 \times 10^{-40}$, which means that the relative error is almost 100 times larger than expected

$$\hat{t} = 1.00 \times 10^{-40} = (1 + \hat{\delta}) t_{\rm exact} \ , \quad \hat{\delta} = 0.99 = 99\epsilon \ !$$

11.4 Inexact

Occurs when the result of an arithmetic operation is inexact. This situation occurs quite often!

11.5 Summary

The IEEE standard response to exceptions is summarized in Table 11.5.

11.6 Example: messy mathematical operations

Homework 17 (test_exceptions) *The following program performs some messy calculations, like division by 0, etc. Run the*
Fortran version:

```
program test_exceptions
```

IEEE Exception	Operation Result
Invalid Operation	NaN
Division by 0	$\pm\infty$
Overflow	$\pm\infty$ (or floating point max)
Underflow	0 or subnormal
Precision	rounded value

Table 4: The IEEE Standard Response to Exceptions

```
real :: a, b, c, d
c = 0.0
d = -0.0
print*, 'c=',c,'  d=',d
a = 1.0/c
print*, a
b = 1.0/d
print*, 'a=',a,'  b=',b
print*, 'a+b=',a+b
print*, 'a-b=',a-b
print*, 'a/b=',a/b
```

```
end program test_exceptions
```

or the

Matlab version:

```
function test_exceptions()
  c = single(0.0);
  d = single(-0.0);
  one = single(1.0);
  fprintf('c = %f, d = %f\n', c, d);
  a = one/c;
  b = one/d;
  fprintf('a = %f, b = %f \n', a, b);
  fprintf('a + b = %f\n', a+b);
  fprintf('a - b = %f\n', a-b);
  fprintf('a / b = %f\n', a/b);
end
```

11.7 Example: overflow

Homework 18 *The following program computes a very large floating point number b in double precision. This is converted to single precision number*

a, and this may result in overflow. Run the program for $p = 0, 0.1, 0.01$. *For which value the single precision overflow occurs? Note that, if you do not see* `Normal End Here !` *and* `STOP` *the program did not finish normally; trapping was used for the floating point exception. If you see them, masking was used for the exception and the program terminated normally.*

Fortran version:

```
program test_overflow
   real :: a, p
   double precision :: b
   print*, 'Please provide p:'
   read*, p
   b = (1.99d0+p)*(2.d0**127)
   print*, b
   a = b
   print*, a
end program test_overflow
```

Matlab version:

```
function test_overflow(p)
  b = double((1.99 + p)*(2^127));
```

```
  fprintf('Double precision value = %f \n', b);
  a = single(b);
  fprintf('Single precision value = %f \n', a);
end
```

11.8 Example: underflow

Homework 19 *The following program computes a small floating point number (2^{-p}) in single and in double precision. Then, this number is multiplied by 2^p. The theoretical result of this computation is 1. Run the code for $p = 120$ and for $p = 150$. What do you see? Does the Fortran90 compiler obey the IEEE standard? Repeat the compilation with the Fortran77 compiler, and run again. Any differences?*

Fortran version:

```
program test_underflow
  real :: a,b
  double precision :: c,d
  integer :: p
  print*, 'Please provide p'
  read*, p
```

```
  c = 2.d0**(-p)
  d = (2.d0**p)*c
  print*, 'Double Precision: ', d
  a = 2.e0**(-p)
  b = (2.d0**p)*a
  print*, 'Single Precision: ', b
end program test_underflow
```

Matlab version:

```
function test_underflow(p)
  c = 2^(-p);
  d = (2^p)*c;
  fprintf('Double precision: %f\n', d);
  a = single(2^(-p));
  b = 2^p*a;
  fprintf('Single Precision: %f\n', b);
end
```

12 Long Summations

Long summations have a problem: since each individual summation brings an error of 0.5 ulp in the partial result, the total result can be quite inaccurate. Fixes:

- compute the partial sums in a higher precision;
- sort the terms first;
- use Kahan's formula.