

to most significant.

<i>Quotients</i>	107	53	26	13	6	3	1	0
<i>Remainders</i>		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.

<i>Fractional</i>	0.625	0.25	0.5	0
<i>Integer</i>		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.

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

1. Convert the following binary numbers to decimal, octal and hexa: 1001101101.0011, 11011.111001;
2. Convert the following hexa numbers to both decimal and binary: 1AD.CF, D4E5.35A;
3. Convert the following decimal numbers to both binary and hexa: 6752.8756, 4687.4231.

2 Memory

The data and the programs are stored in *binary format* in computer's memory. Memory is organized in *bytes*, where **1 byte = 8 binary digits**. In practice we use multiples of byte.

1 Kb	1024 bytes	2^{10} bytes
1 Mb	1024 Kb	2^{20} bytes
1 Gb	1024 Mb	2^{30} bytes

There are several physical memories in a computer; they form a *memory hierarchy*. Note that the physical chips for cache memory use a different technology than the chips for main memory; they are faster, but smaller and more expensive. Also, the disk is a magnetic storage media, a different technology than the electronic main memory; the disk is larger, cheaper but slower.

Memory Type	Size	Access time
Registers	8 bytes	1 clock cycle
Cache, Level 1	126 Kb - 512 Kb	1 ns
Cache, Level 2	512 Kb - 8 Mb	10 ns
Main memory	8 Mb - 2 Gb	60 ns
Hard drive	2 Gb - 40 Gb	10 ms

2.1 Characters in Memory

Characters are letters of the alphabet, both upper and lower case, punctuation marks, and various other symbols. In the ASCII convention (American Standard Code for Information Interchange) one character uses 7 bits. (there are at most $2^7 = 128$ different characters representable with this convention). As a consequence, *each character will be stored in exactly one byte of memory*.

Problem. Implement the following program

```

program test_char
  character a, b
  a='s'
  write(6,*) 'Please input b:'
  read*, b
  write(6,*) a,b
end program test_char

```

Note how characters are declared and initialized. Run the program successfully.

2.2 The Memory Model

When programming, we think of the main memory as a long sequence of bytes. Bytes are numbered sequentially; each byte is designated by its number, called the *address*.

For example, suppose we have a main memory of 4 Gb; there are 2^{32} bytes in the memory; addresses ranging from $0 \dots 2^{32} - 1$ can be represented using 32 bits (binary digits), or (equiv.) by 8 hexa digits.

Suppose we want to store the string “john”. With one character per byte, we need 4 successive memory locations (bytes) for the string. Each memory location has an *address* and a *content*.

Address	Content
1B56AF72	'j'
1B56AF73	'o'
1B56AF74	'h'
1B56AF75	'n'

When we declare a variable, the corresponding number of bytes is reserved in the memory; the name of the variable is just an alias for the address of the first byte in the storage.

3 Representation of Signed Integers

m binary digits (bits) of memory can store 2^m different numbers. They can be positive integers between $\boxed{00 \dots 00} = (0)_{10}$ and $\boxed{11 \dots 11} = (2^m - 1)_{10}$. For example, using $m = 3$ bits, we can represent any integer between 0 and 7.

If we want to represent signed integers (i.e. both positive and negative numbers) using m bits, we can use one of the following methods:

- *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 range between 0 and 3; 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	([S]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 $\bar{x} = x + 2^{m-1} \in [0, 2^m - 1]$. Adding the *bias* 2^{m-1} gives positive results.

Number ₁₀	(biased) ₁₀	(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

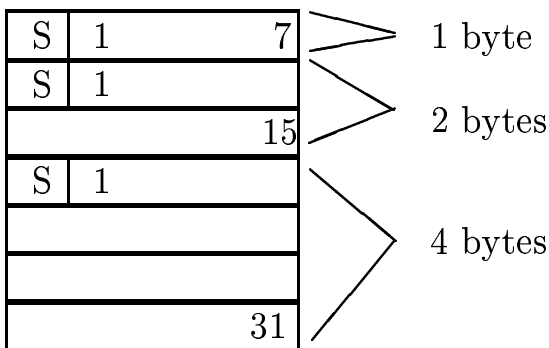
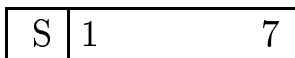
The first bit is 0 for negative numbers, and 1 for nonnegative numbers.

3.1 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	...	$2^{8p} - 1$
Signed integers:	-2^{8p-1}	...	$2^{8p-1} - 1$



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

Problem. Write a Fortran program in which you define two integer variables m and i . Initialize m to 2147483645. Then read i and print out the sum $m + i$.

```

program test_int
  implicit none
  integer :: m,i
  m = 2147483645

```

```

do i=1,10
  print*, 'i=', i, '. m+i=', m+i
end do
end program test_int

```

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 ?

3.2 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).

4 Floating-Point 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 FP. Modern processors have special FP instructions, compilers must generate such FP instructions, and the operating system must handle the exception conditions generated by these FP instructions.

We will now illustrate the floating point representation in base $\beta = 10$. Any decimal number x can be *uniquely* written as

$x = \sigma \cdot m \cdot \beta^e$		
σ	+1 or -1	sign
m	$1 \leq m < \beta$	mantissa
e	integer	exponent

For example

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

If we did not impose the condition $1 \leq m < 10$ 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 $1 \leq m < 10$ is satisfied, we say that the mantissa *is normalized*. Normalization guarantees that

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

Suppose our storage space is limited to 6 decimal digits per FP 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).

$$\boxed{\sigma \mid m_1 m_2 m_3 \mid e_1 e_2}$$

Our example number can be then represented as

$$\underbrace{\boxed{+1}}_{\sigma} \quad \underbrace{\boxed{107}}_m \quad \underbrace{\boxed{+2}}_e$$

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 FP 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).

Most real numbers cannot be exactly represented as floating point numbers. For example, numbers with an infinite representation, like $\pi = 3.141592 \dots$, will need to be “approximated” by a finite-length FP number. In our FP system, π will be represented as

$$\boxed{+ \mid 314 \mid 00}$$

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 FP representation $f\ell(x) = m \cdot \beta^e$ is just an approximation of the real number $x = \mu \cdot \beta^e$ (note that it is possible that the floating point exponent differs by 1 from the number exponent but this is of no consequence for our discussion).

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

$$\delta = \frac{x - f\ell(x)}{x} = \frac{\mu - m}{\mu} .$$

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

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

Another measure for the approximation error is the number of *units in the last place*, or *ulps*. The error in ulps is computed as

$$err = |x - f\ell(x)| \times \beta^{d_m - 1 - e} = |\mu - m| \times \beta^{d_m - 1} .$$

where e is the exponent of $f\ell(x)$ and d_m is the number of digits in the mantissa. To understand this concept we first note that The mantissa m is a d_m digit representation of μ , and therefore the first $d_m - 1$ digits in m and μ coincide. We are concerned with the difference in the last (d_m^{th}) digit. Therefore we first scale the difference by β^{-e} , then by $\beta^{d_m - 1}$ to “ignore” the first $d_m - 1$ digits which are identical.

For our example

$$err = |107.625 - 1.07 \times 10^2| \times 10^{3-1-2} = 0.625 \text{ ulps} .$$

The difference between relative errors corresponding to 0.5 ulps is called the *wobble factor*. If $x - f\ell(x) = 0.5 \text{ ulps}$ and $f\ell(x) = m.mmm \dots m \times \beta^e$, then $x - f\ell(x) = (\beta/2 \times \beta^{-d_m}) \times \beta^e$, and since $\beta^e \leq x < \beta^{e+1}$ we have that

$$\frac{1}{2} \times \beta^{-d_m} \leq \frac{x - f\ell(x)}{x} = 0.5 \text{ ulps} \leq \frac{\beta}{2} \times \beta^{-d_m}$$

If the error is n ulps, the last $\log_\beta n$ digits in the number are contaminated by error. Similarly, if the relative error is δ , the last $\log_\beta (2\delta \times \beta^{1-d_m})$ digits are in error.

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, ..., 9.99. For exponents, two digits will provide the range 00, ..., 99.

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

$$\underbrace{\boxed{+1}}_{\sigma} \quad \underbrace{\boxed{000}}_m \quad \underbrace{\boxed{00}}_e$$

In order to overcome this problem, we need to allow for negative exponents also. We will use a *biased representation*: if the bits e_1e_2 are stored in the exponent field, the actual exponent is $e_1e_2 - 49$ (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

$$\underbrace{\boxed{+1}}_{\sigma} \quad \underbrace{\boxed{123}}_m \quad \underbrace{\boxed{45}}_e$$

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 σ , 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 FP 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 FP 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 FP 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.

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

$$x_{min} = 1.00 \times 10^{-49}, \quad \frac{1}{x_{min}} = 10^{+49} < 9.99 \times 10^{50} = x_{max}.$$

(with a bias of 50 we would have had $1/x_{min} = 10^{50} > 9.99 \times 10^{49} = x_{max}$).

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

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

For example,

$$1101011.101 = +1 \cdot 1.101011101 \cdot 2^6. \quad (1)$$

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

$$\underbrace{\boxed{+1}}_{\sigma} \quad \underbrace{\boxed{110101}}_m \quad \underbrace{\boxed{0110}}_e \quad (2)$$

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.

$$1 \leq x < 2 \rightarrow (x)_2 = \mathbf{1}.m_1m_2m_3\dots$$

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:

$$\underbrace{\boxed{+1}}_{\sigma} \quad \underbrace{\boxed{10101\mathbf{1}}}_{m} \quad \underbrace{\boxed{0110}}_{e} \quad (3)$$

We can now 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). Question: Why do we prefer

5 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.

5.1 Floating Point Types

The standard defines the following FP types:

Single Precision. (4 consecutive bytes/ number).

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

Useful for most short calculations.

Double Precision. (8 consecutive bytes/number)

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

Most often used with scientific and engineering numerical computations.

Extended Precision. (10 consecutive bytes/number).

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

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 $\mathbf{x}*\mathbf{y}$ is correct within 0.5 ulp, and so is the result of $\log(\mathbf{x})$. 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.

5.2 Detailed IEEE representation

(for single precision standard; double is similar)

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

Signum. “ \pm ” bit = 0 (positive) or 1 (negative).

Exponent. Biased representation, with an exponent bias of $(127)_{10}$.

Mantissa. Hidden bit technique.

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

Note that $-e_{min} < e_{max}$, which implies that $1/x_{min}$ does not overflow.

5.3 Number range

The range of numbers represented in different IEEE formats is summarized in Table 5.3.

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

Table 1: IEEE floating point number exponent ranges

5.4 Precision

To define the *precision of the FP system*, let us go back to our toy FP 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 FP system due to the storage of only a finite number of digits.

The precision of the floating point system (the “machine precision”) is the smallest number ϵ for which $1 + \epsilon > 1$.

For our toy FP 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 $\log_{10} p$.

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

Table 2: Precision of different IEEE representations

In (binary) IEEE arithmetic, the first single precision number larger than 1 is $1 + 2^{-23}$, while the first double precision number 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 this yourselves.

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

Remark: 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 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.

Problem. Consider the real number $(0.1)_{10}$. Write its single precision, floating point representation. Does the hidden bit technique result in a more accurate representation?

Problem. What is the gap between 1024 and the first IEEE single precision number larger than 1024?

Problem. 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

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

Problem. 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`).

```

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 program test_precision

```

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.

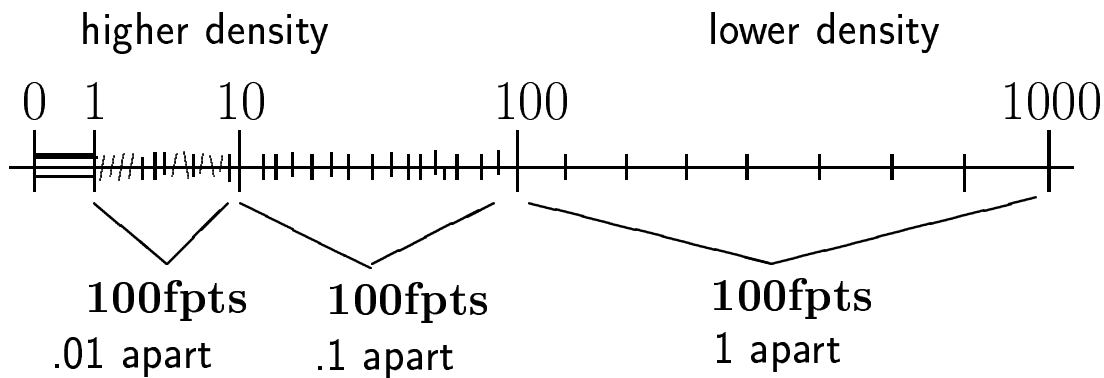
6 The Set of FP Numbers

The set of all FP numbers consists of

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

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

The FP 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.



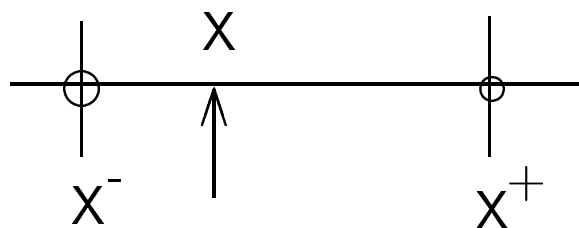
For example, in our FP system, there are 90 points between 1 and 10 (hence, the gap between 2 successive numbers is 0.01). Between 10 and 100 there are again 90 FP numbers, now with a gap of 0.1. The interval 100 to 1000 is “covered” by another 90 FP values, the difference between 2 successive ones being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

In general, if $m \times 10^e$ is a normalized FP number, with mantissa $1.00 \leq m < 9.98$, the very next FP 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 FP 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 $\epsilon \times 2^e$.

7 Rounding - up or down

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 FP numbers X^- and X^+ .



In order to represent X in the computer, we need to approximate it by a FP number. 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 FP number also, but this makes little sense, as the approximation error will be larger than with X^\pm . For example, $\pi = 3.141592\dots$ 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 FP number that approximates X . Then

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

Obviously, when rounding up or down we have to 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 FP candidates can be represented as $X^- = m \times 2^e$ and $X^+ = (m + \epsilon) \times 2^e$ (this is correct since they are successive FP numbers). For now suppose both numbers are positive (if negative, a similar reasoning applies). Since

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

we have

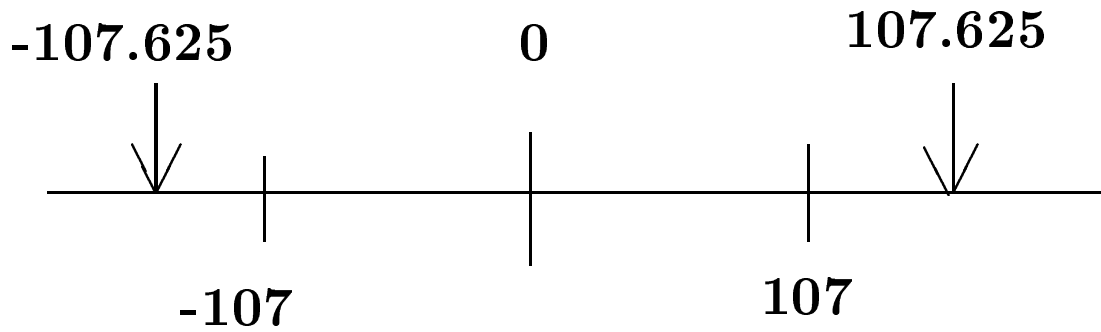
$$|\delta| \leq \frac{|X^+ - X^-|}{X^-} = \frac{\epsilon \times 2^e}{m \times 2^e} \leq \epsilon .$$

Problem. Find an example of X such that, in our toy FP 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.

8 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 FP representation is precisely X^- , and we have $0 \leq X^- < X$. Now, if X was negative, $X = -107.625$, the chopped representation would be $\boxed{-1} \boxed{107} \boxed{+2}$, but now this is X^+ . Note that in this situation $X < X^+ \leq 0$. In consequence, with chopping, we choose X^- if $X > 0$ and X^+ if $X < 0$. 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

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

For example:

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

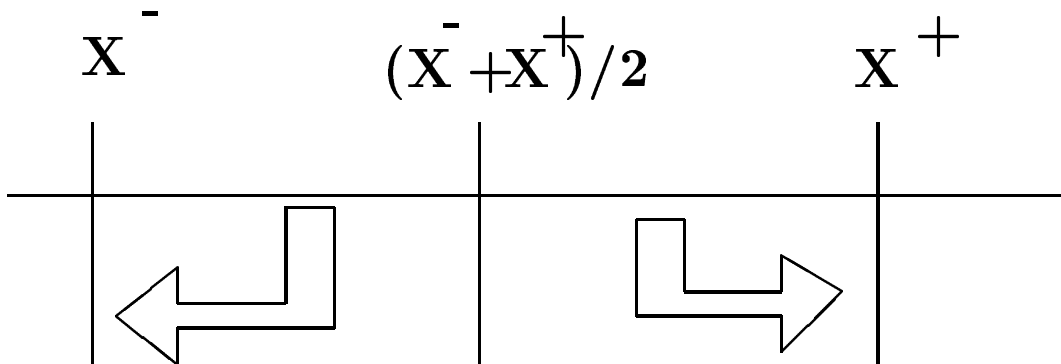
and

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

9 Rounding to nearest.

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

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



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 obtained 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.

Problem. 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”.)

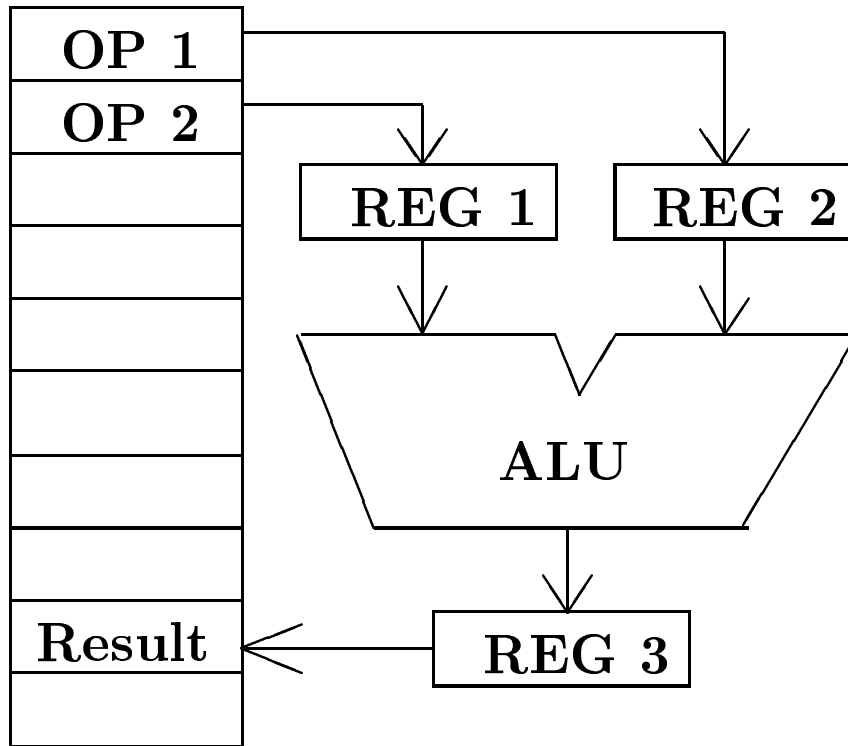
In conclusion, real numbers are approximated and represented in the floating point format. The IEEE standard recognizes four approximation modes:

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.

10 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.



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 FP numbers, the result of an arithmetic operation may not be a FP number.

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

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

The result (103.63) is not a floating number. We can round the result to obtain $\boxed{1.04} \boxed{02}$ (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 $c = \boxed{9.99 \mid -1}$ (0.999). Then

$$(a \oplus b) \oplus c = \boxed{1.04 \mid 02}(104),$$

while

$$a \oplus (b \oplus c) = \boxed{1.05 \mid 02}(105)$$

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

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

11 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,

$$a \oplus b = fl(a + b)$$

$$a \ominus b = fl(a - b)$$

$$a \otimes b = fl(a \times b)$$

$$a \oslash b = fl(a/b).$$

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, $2.0/3.0 = 0.666\dots$).

12 The Guard Digit

Is useful when subtracting almost equal numbers. Suppose $a = (1.0)_2 \times 2^0$ and $b = (1.11\dots 1)_2 \times 2^{-1}$, with 23 1's after the binary point. Both a and b are single precision floating point numbers. The mathematical result is $a - b = (1.0)_2 \times 2^{-24}$. It is a floating point number also, hence the numerical result should be identical to the mathematical result, $a \ominus b = fl(a - b) = a - b$.

When we subtract the numbers, we align them by shifting b one position to the right. If computer registers are 24-bit long, then we may have one of the following situations.

1. Shift b and “chop” it to single precision format (to fit the register), then subtract.

$$\begin{array}{r} 1.000 \dots 000 \\ - 0.111 \dots 111 \\ \hline 0.000 \dots 001 \end{array}$$

The result is 2^{-23} , twice the mathematical value.

2. Shift b and “round” it to single precision format (to fit the register), then subtract.

$$\begin{array}{r} 1.000 \dots 000 \\ - 1.000 \dots 000 \\ \hline 0.000 \dots 000 \end{array}$$

The result is 0, and all the meaningful information is lost.

3. Append the registers with an extra **guard bit**. When we shift b , the guard bit will hold the 23^{rd} 1. The subtraction is then performed in 25 bits.

$$\begin{array}{r} 1.000 \dots 000 \ [0] \\ - 0.111 \dots 111 \ [1] \\ \hline 0.000 \dots 000 \ [1] \end{array}$$

The result is normalized, and is rounded back to 24 bits. This result is 2^{-24} , precisely the mathematical value. Funny fact: Cray supercomputers lack the guard bit. In practice, many processors do subtractions and additions in extended precision, even if the operands are single

or double precision. This provides effectively 16 guard bits for these operations. This does not come for free: additional hardware makes the processor more expensive; besides, the longer the word the slower the arithmetic operation is.

The following theorem (see David Goldberg, p. 160) shows the importance of the additional guard digit. *Let x, y be FP numbers in a FP system with β, d_m, d_e ;*

- *if we compute $x - y$ using d_m digits, then the relative rounding error in the result can be as large as $\beta - 1$ (i.e. all the digits are corrupted!).*
- *if we compute $x - y$ using $d_m + 1$ digits, then the relative rounding error in the result is less than 2ϵ .*

Note that, although using an additional guard digit greatly improves accuracy, it *does not* guarantee that the result will be exactly rounded (i.e. will obey the IEEE requirement). As an example consider $x = 2.34 \times 10^2$, $y = 4.56$ in our toy FP system. In exact arithmetic, $x - y = 229.44$, which rounds to $fl(x - y) = 2.29 \times 10^2$. With the guard bit arithmetic, we first shift y and chop it to 4 digits, $\hat{y} = 0.045 \times 10^2$. Now $x - \hat{y} = 2.295 \times 10^2$ (calculation done with 4 mantissa digits). When we round this number to the nearest (even) we obtain 2.30×10^2 , a value different from the exactly rounded result.

However, by introducing a second guard digit and a third, “sticky” bit, the result is the same as if the difference was computed exactly and then rounded (D.Goldberg, p. 177).

13 Special Arithmetic Operations

13.1 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 $1/x \neq 1/y$!

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

Table 3: IEEE results to comparisons

13.2 Operations with ∞

The following operations with infinity are possible:

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

13.3 Operations with NaN

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

13.4 Comparisons

The IEEE results to comparisons are summarized in Table 13.4.

14 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 FP 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.

14.1 Division by 0

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

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

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.

14.2 Overflow

Occurs when the result of an arithmetic operation is finite, but larger in magnitude than the largest FP 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 FP 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 FP 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" FP 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

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

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

$$\frac{x^2}{7}(x + 1.0 \times 10^{23})^2 = 9.93 \times 10^{-1}$$

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.

14.3 Underflow

Occurs when the result of an arithmetic operation is smaller than the smallest *normalized* FP 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

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

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 $|\delta_{\otimes}| \leq \epsilon/2$:

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

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

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

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

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

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

Table 4: The IEEE Standard Response to Exceptions

14.4 Inexact

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

14.5 Summary

The IEEE standard response to exceptions is summarized in Table 14.5.

15 Flags and Exception Trapping

Each exception is signaled by setting an associate status flag; the flag remains set until explicitly cleared. The user is able to read and write the status flags. There are 5 status flags (one for each possible exception type); in addition, for each flag there is a trap enable bit (see below), and there are 4 rounding modes bits. If the result of a computation is, say, $+\infty$, the flag values help user decide whether this is the effect of an overflow or is a genuine infinity, like $1/0$.

The programmer has the option to

- *Mask the exception.* The appropriate flag is set and the program continues with the standard response shown in the table;
- *Trap the exception.* Occurrence of the exception triggers a call to a special routine, the trap handler. Trap handlers
 - receive the values of the operands which lead to the exception, or the result;
 - clear or set the status flag; and
 - return a value that will be used as the result of the faulty operation.

Using trap handler calls for each inexact operation is prohibitive.

For overflow/underflow, the argument to the trap handler is the result, with a modified exponent (the "wrapped-around" result). In single precision the exponent is decreased/increased by 192, and in double precision by 1536, followed by a rounding of the number to the corresponding precision.

Trap handlers are useful for backward compatibility, when an old code expects to be aborted if exception occurs. Example (from D. Goldberg, page 189): without aborting, the sequence

```
doSuntil(x >= 100)
```

will loop indefinitely if x becomes NaN.

16 System Aspects

The design of computer systems requires in-depth knowledge about FP. Modern processors have special FP instructions, compilers must generate such FP instructions, and the operating system must handle the exception conditions generated by these FP instructions.

16.1 Instruction Sets

It is useful to have a multiplication of single precision operands (p mantissa digits) that returns a double precision result ($2p$ mantissa digits). All calculations require occasional bursts of higher precision.

16.2 Ambiguity

A language should define the semantics precisely enough to prove statements about the programs. Common points of ambiguity:

- $x=3.0/10.0$ FP number, it is usually not specified that all occurrences of $10.0*x$ must have the same value.
- what happens during exceptions.
- interpretation of parenthesis.
- evaluation of subexpressions. If x real and m,n integers, in the expression $x+m/n$ is the division integer or FP? For example, we can compute all the operations in the highest precision present in the expression; or we can assign from bottom up in the expression

graph tentative precisions based on the operands, and then from top down assign the maximum of the tentative and the expected precision.

- defining the exponential consistently. Ex: $(-3)**3 = -27$ but $(-3.0)**(3.0)$ is problematic, as it is defined via logarithm. Goldberg proposes to consider $f(x) \rightarrow a, g(x) \rightarrow b$ as $x \rightarrow 0$. If $f(x)^{g(x)} \rightarrow c$ for all f, g then $a^b = c$. For example, $2^\infty = \infty$, but $1^\infty = NaN$ since $1^{1/x} \rightarrow 1$ but $(1-x)^{1/x} \rightarrow e^{-1}$.

16.3 Programming Language Features

The IEEE standard says nothing about how the features can be accessed from a programming language. There is usually a mismatch between IEEE-supporting hardware and programming languages. Some capabilities, like exactly rounded square root, can be accessed through a library of function calls. Others are harder:

- The standard requires extended precision, while most languages define only single and double.
- Languages need to provide subroutines for reading and writing the state (exception flags, enable bits, rounding mode bits, etc).
- Cannot define $-x = 0 - x$ since this is not true for $x = +0$;
- NaN are unordered, therefore when comparing 2 numbers we have $<, >, =, unordered$.
- The precisely defined IEEE rounding modes may conflict with the programming language's implicitly-defined rounding modes or primitives.

16.4 Optimizers

Consider the following code for estimating the machine ϵ

```
eps = 1.0; doeps = 0.5 * eps; while(eps + 1 > 1);
```

If the compiler "optimizes" ($\mathit{eps} + 1 > 1$) to ($\mathit{eps} > 0$) the code will compute the largest number which is rounded to 0.

Optimizers should be careful when applying mathematical algebraic identities to FP variables. If, during the optimization process, the expression $x + (y + z)$ is changed to $(x + y) + z$, the meaning of the computation is different.

Converting constants like $1.0E-40*x$ from decimal to binary at compile time can change the semantic (a conversion at run time obeys the current value of the IEEE rounding modes, and eventually raise the inexact and underflow flags).

Semantics can be changed during common subexpression elimination. In the code

$$C = A * B; RndMode = Up; D = A * B;$$

$A * B$ is not a common subexpression, since it is computed with different rounding modes.

16.5 Exception Handling

When an operation traps, the conceptual model is that everything stops and the trap handler is executed; since the underlying assumption is that of serial execution, traps are harder to implement on machines that use pipelining or have multiple ALU. Hardware support for identifying exactly which operation did trap may be needed. For example,

$$x = y * z; z = a + b;$$

both operations can be physically executed in parallel; if the multiplication traps, the handler will need the value of the arguments y and z . But the value of z is modified by the addition, which started in the same time as the multiply and eventually finished first. IEEE supporting systems must either avoid such a situation, or provide a way to store z and pass the original value to the handler, if needed.

17 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.

18 Typical pitfalls with floating point programs

All numerical examples in this section were produced on an Alpha 21264 workstation. On other systems the results may vary, but in general the highlighted problems remain the same.

18.1 Binary versus decimal

Consider the code fragment

```
program test
  real :: x=1.0E-4
  print*, x
end program test
```

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.

18.2 Floating point comparisons

Because of the inexactities, it is best to avoid strict equality when comparing floating point numbers. For the above example, the code

```
if ( (1.0E+8*x**2) == 1.0 ) then
  print*, 'Correct'
end if
```

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:

```
epsilon = 1.0E-7
w = 1.0E+8 * x**2
if ( abs(w-1.0) .LE. 0.5*epsilon*( abs(w)+abs(1.0) ) ) then
  print*, 'Correct'
end if
```

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

In the following example the branch correct is taken:

```
program quiz_2b
  implicit none
  real :: x
```

```

x = 1.0/2.0
if ( (2.0*x) .eq. 1.0 ) then
  print*, 'Correct'
else
  print*, 'Funny'
end if
end program quiz_2b

```

while in the next the branch incorrect is taken:

```

program quiz_2a
  implicit none
  real :: x
  x = 1.0/3.0
  if ( (3.0*x) .eq. 1.0 ) then
    print*, 'Correct'
  else
    print*, 'Funny'
  end if
end program quiz_2a

```

18.3 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
  real :: x = 1.0E-4
  integer :: i
  i = 10000*x
  print *, i
end program test

```

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

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

```

program test
  real :: x = 1.234567

```

```

double precision :: y = 0.0D0
y = x
print *, 'X =',x,' Y =',y
end program test

```

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 inexactness 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.

18.4 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 FP registers for Pentium are in extended precision, 80 bits) while for `c` the operands `a` and `b` are stored in the memory. (note that the Alpha compiler produces zero).

18.5 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

$$ax^2 + bx + c = 0 ,$$

where all the coefficients are FP numbers

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

using our toy decimal FP 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 FP 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%!

The same happens in single precision:

```
real :: a=1.0, b=-1.0E+8, c=9.999999E+7
d = sqrt(b**2-4.0*a*c)
r1 = (-b+d)/(2.0*a)
r2 = (-b-d)/(2.0*a)
e2 = (2.0*c)/(-b+d)
```

The exact results are -1 and $-c$, and we expect the numerical results to be close approximations. We have $b**2 = 1.0E + 16$ and $4ac = 3.9999997E + 08$; due to cancellation errors the computed value of d is $d = 1.0E + 8$. Then $r1 = 0$ and $r2 = 1.0E + 8$.

With

```
a=1.0E-3,b=-9999.999,c=-1.0E+4
```

The exact results are -1 and $1.0E + 7$. d is calculated to be $1.d + 4$, and $b - d$ suffers from cancellation errors. The numerical roots are $1.0E + 07$ (exact!) and -0.9765624 (about 2.5% relative error, much higher than the expected $1.0e - 7!$).

To overcome this, we might avoid the cancellation by using mathematically equivalent formulas:

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

With this formula, $r_2 = (2c)/(-2) = -9.99 \times 10^{-1}$, a much better approximation. For the second example $e2 = 0.9999999$. For the third example the root is -1 (exact).

18.6 Insignificant Digits

Consider the Fortran code

```

program test
  real :: x=100000.0, y=100000.1, z
  z = y-x
  print*, 'z=',z
end program test

```

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!

18.7 Order of Operations Matters

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

```

program test
  real :: x=12345.6, y=45678.9, z=98765432.1
  real :: w1, w2
  w1 = x*y/z
  w2 = y*(x*(1.0/z))
  print*, w1-w2
end program test

```

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

19 Integer Multiplication

As another example, consider the multiplication of two single-precision, FP 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.

20 Homework

Problem. The following program computes a very large FP number in double precision. When assigning $a=b$, the double precision number (b) will be converted to single precision (a), and this may result in overflow. Compile the program with both Fortran90 and Fortran77 compilers. For example, `f90 file.f -o a90.out` and `f77 file.f -o a77.out`. Run the two versions of the program for $p = 0, 0.1, 0.01$. Does the Fortran90 compiler obey the IEEE standard? For which value the single precision overflow occurs? How about the Fortran77 compiler? Note that, if you do not see `Normal End Here !` and `STOP` the program did not finish normally; trapping was used for the FP exception. If you see them, masking was used for the exception and the program terminated normally. Do the compilers use masking or trapping?

```

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
  print*, 'normal end here !'
end program test_overflow

```

Problem. The following program computes a small FP 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?

```
%
  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
    print*, 'normal end here !'
  end program test_underflow
```

Problem. The following program performs some messy calculations, like division by 0, etc. Compile it with both Fortran90 and Fortran77 compilers, and run the two executables. What do you see? Any differences?

```
program test_arit
  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_arit
```