Storing Numbers in CPUs

Enrico Calore

enrico.calore@fe.infn.it

Sebastiano Fabio Schifano

sebastiano.schifano@unife.it

University of Ferrara and INFN

Università
degli Studi
di Ferrara

Introduction

Introduction

The relevance of what we are discussing depends on the field of applications

- multimedia, video-games
- analysis of scientific data
- encryption of data
- ...

And in general is important to know that for computers

- $a + b \neq b + a$
- $a - b = 0 \not\Rightarrow a = b$
- ...

Binary System

Data Storing

Data are represented using only two symbols: **0** and **1**.

This because it is easy to build electronic devices that represents two states.

The smallest unit is called **bit** shorthand for **binary digit**.

The **Byte** is a sequence of 8 bits, e.g $1101\ 0010$

The **Word** is a sequence of bytes, e.g. 2, 4 or 8 corresponding to 16, 32 and 64 bits.

The Base 10 System

In the base 10 system each number is represented as a sequence of symbols

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

The **value** associated to a sequence of n symbols

$$c_{n-1}\ c_{n-2}\ \dots\ c_2\ c_1\ c_0$$

is given by

$$c_{n-1} \cdot 10^{n-1} + c_{n-2} \cdot 10^{n-2} \dots c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0 \cdot 10^0$$

- the notation is **positional** meaning that each symbol has a **weight** corresponding to the position of the symbol in the sequence
- the **weights** are powers of the **base** with the exponent index corresponding to the position of the symbol in the sequence starting from right and counting from zero

Examples:

- $57 = 5 \cdot 10^1 + 7 \cdot 10^0$
- $147 = 1 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$

The Base 2 System

At the same way, in the **base** 2 system numbers are representd by a sequence of 0 and 1.

The **value** corresponding to a sequence of n bits or symbols $c_{n-1}\ c_{n-2}\ \ldots\ c_2\ c_1\ c_0$ is given by

$$c_{n-1} \cdot 2^{n-1} + c_{n-2} \cdot 2^{n-2} \ldots c_2 \cdot 2^2 + c_1 \cdot 2^1 + c_0 \cdot 2^0$$

- the notation is **postional** meaning that each symbol has a different **weight** corresponding to the position of the symbol in the sequence
- the **weights** are powers of the **base** with the exponent index corresponding to the position of the symbol in the sequence starting from right and counting from zero

Examples:

- $1 = 1 \cdot 2^0 = 1_{10}$
- $10 = 1 \cdot 2^1 + 0 \cdot 2^0 = 2_{10}$
- $101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{10}$
- $1111 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 15_{10}$

Converting numbers from base 10 to base 2

If X_{10} is a number in base 10, we may get the sequence of bits representing the number in base 2:

$$c_{n-1}\ c_{n-2}\ \ldots\ c_1\ c_0$$

using the following **alghoritm**

1. let $i = 0$
2. compute the quotient q_i and the remainder r_i dividing $\frac{X}{2}$, the make the assignment $X = 2 \cdot q_i + r_i$
3. let $c_i = r_i$
4. **if** $q_i = 0$ **then** stop, **else** make $X = q_i$ and $i = i + 1$, and go back to step 2

the sequence of $c_i \in \{0, 1\}$ is the binary representation of X.

An algorithm is a sequence of well defined steps that solve a problem.

Converting from base 10 to base 2

Example: converting 13 to base 2

- $13/2 = 6 \cdot 2 + 1 \rightarrow c_0 = 1$
- $6/2 = 3 \cdot 2 + 0 \rightarrow c_1 = 0$
- $3/2 = 1 \cdot 2 + 1 \rightarrow c_2 = 1$
- $1/2 = 0 \cdot 2 + 1 \rightarrow c_3 = 1$

binary representation is $13_{10} = 1101$.

Example: converting 16 to base 2

- $16/2 = 8 \cdot 2 + 0 \rightarrow c_0 = 0$
- $8/2 = 4 \cdot 2 + 0 \rightarrow c_1 = 0$
- $4/2 = 2 \cdot 2 + 0 \rightarrow c_2 = 0$
- $2/2 = 1 \cdot 2 + 0 \rightarrow c_3 = 0$
- $1/2 = 0 \cdot 2 + 1 \rightarrow c_4 = 1$

binary representation is $16_{10} = 1\,0000$.

Fractions in Binary

To write the binary representation of a rational number $i.d$, with i being the integer part and d the decimal part, we convert i and d separetely.

To convert the decimal part $d < 1$, we have to write a sequence of binary digits $c_{-1}\ c_{-2}\ c_{-3}\ \ldots$ such that d can be expressed ad sum of negative powers of 2:

$$d = \sum_{h=1}^{n} c_{-h} \cdot 2^{-h}$$

To extract the sequence $c_{-1}\ c_{-2}\ c_{-3}\ \ldots$ we may use the following algorithm:

1. let $i = 1$
2. compute $p = d \cdot 2$;
3. if $p < 1$, then the corrisponding binary digit is $c_{-i} = 0$, equal $d = p$ and go back to step 2;
4. if $p > 1$, then the corresponding binary digit is $c_{-i} = 1$, equal $d = p - 1$, and go back to step 2;
5. if $p = 1$, then $c_{-i} = 1$ and **STOP**.

The sequence $c_{-i} \in \{0, 1\}$ is the binary representation of the number.

Fractions in Binary

Example: find the binary representation of 0.125

- $0.125 \cdot 2 = 0.25 \rightarrow c_{-1} = 0$
- $0.25 \cdot 2 = 0.5 \rightarrow c_{-2} = 0$
- $0.5 \cdot 2 = 1.0 \rightarrow c_{-3} = 1$

then $0.125_{10} = 0.001$ corresponds to

$$0.125 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 \cdot 0.125 = 0.125$$

Example: find the binary representation of 0.6875

- $0.6875 \cdot 2 = 1.375 \rightarrow c_{-1} = 1$
- $0.375 \cdot 2 = 0.75 \rightarrow c_{-2} = 0$
- $0.75 \cdot 2 = 1.5 \rightarrow c_{-3} = 1$
- $0.5 \cdot 2 = 1.0 \rightarrow c_{-4} = 1$

then $0.6875_{10} = 0.1011$ correponds to

$$0.6875 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 1 \cdot 0.5 + 1 \cdot 0.125 + 1 \cdot 0.0625 = 0.6875$$

Fractions in Binary

Example: find the binary representation of 0.6

- $0.6 \cdot 2 = 1.2 \rightarrow c_{-1} = 1$
- $0.2 \cdot 2 = 0.4 \rightarrow c_{-2} = 0$
- $0.4 \cdot 2 = 0.8 \rightarrow c_{-3} = 0$
- $0.8 \cdot 2 = 1.6 \rightarrow c_{-4} = 1$
- $0.6 \cdot 2 = 1.2 \rightarrow c_{-5} = 1$
- $0.2 \cdot 2 = 0.4 \rightarrow c_{-6} = 0$
- $0.4 \cdot 2 = 0.8 \rightarrow c_{-7} = 0$
- $0.8 \cdot 2 = 1.6 \rightarrow c_{-8} = 1$
- . . . the sequence is repeating since the number is periodic

then $0.6_{10} = 0.\overline{1001}$.

If the sequence of binary digits is repeating then the binary representation of the number is **periodic** even if this was not true in base 10.

Converting from base 2 to base 10

The base 10 representation of a binary number

$$c_{n-1}\ c_{n-2}\ \ldots\ c_2\ c_1\ c_0 \bullet c_{-1}\ c_{-2} \ldots c_{-h}$$

can be extract as addition of

- positive powers of two for the integer part
- negative powers of two for the decimal part

$$\sum_{i=-h}^{n-1} = c_i \cdot 2^i$$

Example: 101011.1011 in base 10 is

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 43.6875$$

Binary Representation of Negative Numbers

Negative numbers can be represented putting a minus sign symbol at the end of the leftmost digit of the binary representation of the absolute value of the number.

Examples:

- $-7 = -111$
- $-15.0625 = -1111.0001$
- $-43.6875 = -101011.1011$
- ...

Arithmetic Operations

The arithmetic operations in base 2 can be done applying the same rules we use for base 10.

- to multiply a number by 2^n
  - move the digits to the left by n positions inserting zeros to the right:

    Example: $100_2 \cdot 100_2 = 1\,0000_2 = 4_{10} \cdot 4_{10} = 16_{10}$
  - if the number is a fraction, move the radix point by n positions to the right adding zeros if necessary:

    Example:: $1011 \cdot 1000 = 1011000$, in base 10 correspond to the operation $11 \cdot 8 = 88$
- to divide a number by 2^{-n},
  - move the digits to the right by n positions inserting zeros to the left:

    Example: $1011 : 10 = 10 = 11_{10} : 4_{10} = 2_{10}$
  - if the number is a fraction move the radix point by n positions to the left inserging zeros if necessary:

    Example: $111 : 10 = 11.1$, in base 10 correspond to the operation $7 : 2 = 3.5$.

Operazioni aritmetiche

For addition we may apply the same rules of base 10 keeping in mind that $1 + 1 = 0$.

- the addition can be computed summing the corresponding digits
- accounting that $1 + 1 = 0$ with carry 1

Example

- $1111 + 11$ corresponding in base 10 to $15 + 3$

$$\begin{array}{rl} 1111 & + \\ 11 & = \\ \hline 10010 & \end{array}$$

- $1001101.1011 + 1101100.1101$ corresponding in base 10 to $77.6875 + 108.8125$

$$\begin{array}{rl} 1001101.1011 & + \\ 1101100.1101 & = \\ \hline 10111010.1000 & \end{array}$$

Octale and Exadecimal Base

In computer science is also common to use octal and hexadecimal bases.

- octal and hexadecimal are positional notations
- in the octal base we use the symbols

$$0, 1, 2, 3, 4, 5, 6, 7$$

  and each symbol can be stored as a semi-byte corresponding to 4 bits;
- in the hexadecimal base we use the symbols

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$$

  and each symbol can be stored as a byte corresponding to 8 bits.

Examples:

- $573.671_8 = 5 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 + 6 \cdot 8^{-1} + 7 \cdot 8^{-2} + 1 \cdot 8^{-3} = 379.861328125_{10}$
- $97A.6E1_{16} = 9 \cdot 16^2 + 7 \cdot 16^1 + 10 \cdot 16^1 + 6 \cdot 16^{-1} + 14 \cdot 16^{-2} + 1 \cdot 16^{-3} = 2576.429931640625_{10}$

Binary, Octal and Hexadecimal Numbers

- binary notation is commonly used at level of hardware
- octal and hexadecimal notations are commonly used at level of software to have a compact representation of binary numbers.

base 10	base 2	base 8	base 16
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Storing Numbers

Storing Numbers

Computers store data using **binary** representation with a **finite** number of digits.

- sequences of N bits are used with the convention that the most significant – usually the leftmost – has weight 2^{N-1} and the least significant – usually the rightmost – has weight 2^0
- numbers of different types are stored using different representations.

We use the terms:

- **LSB**="least significant bit", **MSB**"most significant bit"
- **Byte** sequence of 8 bit
- **Word** as a sequence if 2, 4, 8 Bytes (this is machine dependent) and corresponds to the smallest chunck data read/write from/to memory.

Storing Natural Numbers

Storing Natural Numbers

- N bits are used, commonly $N = 32$ but it is also possibile to have $N = 16$ and $N = 64$
- using N bits we may represents 2^N numbers in the interval $[0 \dots 2^N - 1]$
  - $N = 4$, we represent $2^4 = 16$ numbers in the range $[0 \dots 15] = [0 \dots 2^4 - 1]$
  - $N = 16$, we represent $2^{16} = 65536$ numbers in the range $[0 \dots 65535] = [0 \dots 2^{16} - 1]$
  - $N = 32$, we represent $2^{32} = 4\,294\,967\,296$ numbers in the range $[0 \dots 4294967295] = [0 \dots 2^{32} - 1]$
  - $N = 64$, we represent $2^{64} = 18\,446\,744\,073\,709\,551\,616$ numbers in the range

$$[0 \dots 18446744073709551615] = [0 \dots 2^{64} - 1]$$

Storing Natural Numbers

Multiplying or adding two N bits numbers we may get a result that can not fit into N bits generating a condition called **overflow**:

- $N = 4,\ 1000_2 + 1000_2 = 10000_2$, in decimale $8 + 8 = 16$
- $N = 4,\ 0111_2 \cdot 0100_2 = 11100_2$, in decimale $7 \cdot 4 = 28$

Adding two numbers the **overflow** can be generated as carry over the MSB bits column.

Example: assume $N = 4$ bits:

$$\begin{array}{rl} 1110 & + \\ 0010 & = \\ \hline 10000 & \end{array}$$

the results can not be represented using $N = 4$ bits.

Storing Integer Numbers

Storing Integer Numbers

Integer numbers can be stored using the represention called **sign and magnitude**:

- one bit – usually the MSB – is used to store the sign: 0 meaning **positive**, 1 meaning **negative**
- the other bits are used to represent the magnitude or absolute value of the number in base 2

Example: if $N = 8$, the sequence

$$\mathbf{1}0001011$$

represents the number -11_{10}, and the bit 7 is the sign bit.

Using this representation with N bits we may store all numbers in the range

$$[\ -(2^{N-1} - 1)\ \ldots\ +(2^{N-1} - 1)\ ]$$

Example: if $N = 10$, the range of numbers is

$$[-511\ \ldots\ +511]$$

Storing Integer Numbers: Issues

Using the representation sign-and-magnitude the **zero** has two configurations, the **zero positive** and the **zero negative**.

Moreover, computation of the sign bit for additions and subtractions is not straightforward; for example assume to have $N = 8$:

- $a = 00011011 = 27_{10}$, $b = 00101011 = 43_{10}$, $a - b = 10010000$
- $a = 00111011 = 59_{10}$, $b = 00101011 = 43_{10}$, $a - b = 00010000$

then, to compute the sign bit we need to check the absolute value of operands, if $|b| > |a|$ the sign bit is negative, else it is positive.

Example: $N = 4$

Numeri positivi		Numeri positivi	
0000	+0	1000	-0
0001	+1	1001	-1
0010	+2	1010	-2
0011	+3	1011	-3
0100	+4	1100	-4
0101	+5	1101	-5
0110	+6	1110	-6
0111	+7	1111	-7

For these reasons the representation sign-and-magnitude is not used.

Representation of Ingers using Two's Complement Notation

The representation **two's complement** using N bits (es. $N = 32$) allows to represent the numbers z in the asymetric range $[-2^{N-1} \dots 2^{N-1} - 1]$.

- if N are the bits available
- if $z \in [-2^{N-1} \dots 2^{N-1} - 1]$ is the integer number to store
- the number is stored coding on N bits the value

$$z_{2C} = 2^N - |z|$$

calles **two's complement** of z on N bits

- subtraction is computed on N bits, assuming to borrow a bit from the N^{th} column to compute subtraction of bits of column index $N - 1$ if necessary

Representation of Ingers using Two's Complement Notation

If the number z to store is **positive**

- we code the binary representation of the number over $N-1$ bits
- leaving the MSB set to 0

Example:

- if $N=8$ and $z_{10}=13_{10}$, $z_{2C}=13_{10}=0000\ 1101$
- if $N=4$ and $z_{10}=6_{10}$, $z_{2C}=13_6=0110$
- if $N=4$ and $z_{10}=15_{10}$,
  in this case the number can NOT be represented and we get a **wrong** encoding:
  $z_{2C}=15_{10}=0111$

The largest positive number that we can represent is $z_M=2^{N-1}-1$, e.g. using $N=8$ we have $z_M=127$, abd with $N=4$ we have $z_M=7$.

Representation of Ingers using Two's Complement Notation

If the number z to store is **negative**

- the number is represented using the **two complement** over N bits
- to get the two complement binary representation z_{2C} of z we compute

$$z_{2C} = (2^N - |z|)_2$$

- doing this the MSB (bit $N-1$) is set to 1 since the value of the minuend on N bits is zero and the subtraction sets the bit index $N-1$ to 1.

Example, se $N = 8$, $z = -13_{10}$, we have

$$(-13)_{2C} = 2^8 - |z| = 256 - 13 = 243 = 1111\ 0011$$

since

$1\,0000\,0000$	-	256_{10} -
$0000\,1101$	=	13_{10} =
$\mathbf{1}111\,0011$		243

Representation of Ingers using Two's Complement Notation

Using the two's complement notation

- the largest positive number we can represent is $z_M = 2^{N-1} - 1$
- the smallest negative number we can represent is $z_m = -2^{N-1}$
- using N bits we may represnt numbers in the range $[-2^{N-1} \dots 2^{N-1} - 1]$

Esempio:

- using $N = 4$ we represent numbers in the range $[-2^3 \dots 2^3 - 1] = [-8 \dots 7]$
- using $N = 8$ we represent numbers in the range $[-2^7 \dots 2^7 - 1] = [-128 \dots 127]$
- using $N = 10$ we represent numbers in the range $[-2^9 \dots 2^9 - 1] = [-512 \dots 511]$

Representation of Ingers using Two's Complement Notation

The two's complemet binary representation of a number z can be computed using two different algorithms.

Let z the number and $|z|_2$ the binary representation of absolute value of z:

- Algorithm 1: starting from the LSB e moving towards the MSB of $|z|_2$
  - leave unchanged all bits up to the first 1 included
  - swap all the others bit up to the end (1 becoms 0, and 0 becomes 1)
- Algorithm 2: let z the and $|z|_2$ the binary representation of absolute value of z:
  - swap all the bits of $|z|_2$ getting the **ones' complement representation**,
  - sum 1

Representation of Ingers using Two's Complement Notation

Example: $N = 8$ and $z = -33$

- $|z| = 33 = 0010\,0001$
- applying the first algorithm the two's complement of $|z|$ is

$$1101\,1111$$

- applying the latter algorithm the ones's complement of $|z|$ is

$$1101\,1110$$

and summing 1 we get

$$1101\,1110 + 1 = 1101\,1111$$

Using $N = 8$, the numbers $z \notin [-128 \ldots 127]$ can NOT be represented.

Representation of Ingers using Two's Complement Notation

Esempio: $N = 4$ e $z = -6$

- $|z| = 6 = 0110$
- using the first algorithm we get

$$1010$$

- using the latter algorithm the ones' complement of $|z|$ is

$$1001$$

and adding 1 we get

$$1001 + 1 = 1010$$

Using $N = 4$, the numbers $z \notin [-8 \dots 7]$ can NOT be represented.

Representation of Integers using Two's Complement Notation

WARNING

Computation of complement two depend on the number N of bits used in the representation.

This means that computing the two's complement of a number z requires to apply the algorithm 1 and 2 representing the $|z|$ exactely on N bits.

Example: $N = 8$, $z = -3$

- writing $|z| = 11$, the two's complement is $z_{2C} = 01$, and representing on 8 bits we have $0000\,0001$ **WRONG !**, the sign bit is zero !
- writing $|z| = 0000\,0011$ the two's complement is $z_{2C} = 1111\,1101$, and in this case the sign bit is 1 as expected.

WARNING

Using N bit we represent numbers $z \in [-2^{N-1} \dots 2^{N-1} - 1]$.

Representation of Integers using Two's Complement Notation

Example, using $N = 4$, numbers are coded as:

Positive Numbers		Negative Numbers	
0000	+0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

- only one representation for zero
- numbers that can be represented are in the range $[\ -2^{N-1}\ \ldots\ (2^{N-1}-1)\ ]$, for example with $N = 4$ all numbers in the range $[-8 \ldots 7]$

Converting from two's complement to base 10

To convert a binary number represented using the two's complement notation into the corresponding base 10 number

- if the number is **positive** use the usual converion from binary to base 10
- if the number is **negative**
  - compute the two's complement of the number
  - convert the number into base 10
  - add minus sign

Example:

- $N = 4$, $0111 = 7$
- $N = 4$, 1101 the two's complement is $0011 = 3$, then the result is $1101 = -3$
- $N = 8$, $1101\,0101$ the two's complement is $0010\,1011 = 43$, then the result is $1101\,0101 = -43$

Subtraction

Let us compute $a - b$:

- write the subtrahend $-b$ into two's complement representation
- compute the **sum** $(a)_2 + (-b)_{2C}$ neglecting the carry on the column of MSB bits.

Examples:

- $a = 0001\,1011 = 27_{10}$, $b = 0010\,1011 = 43_{10}$
  - $-43 = 1101\,0101$
  - $a + (-b) = 0001\,1011 + 1101\,0101 = 1111\,0000 = -16$
- $a = 0011\,1011 = 59_{10}$, $b = 0010\,1011 = 43_{10}$
  - $-43 = 1101\,0101$
  - $a + (-b) = 0011\,1011 + 1101\,0101 = 0001\,0000 = 16$

Using the two's complement the sign bit is computed as the others value bit.

Addition

The sum of either two positive integers or negative integers may generate an integer that can not be represented generating an **overflow**.

- if addends have different sign bits no **overflow** can be generated and the result can be represented;
- if addends have the same sign bit
  - if the result have the bit sign equal to the addends the result is correct;
  - otherwise the result is wrong generating an **overflow**.

Esempio: sia $N = 4$

- $a = 1000 = -8_{10}$, $a + a = 1000 + 1000 = 0000$, generate an **overflow** as the result should be positive
- $a = 0111 = 7_{10}$, $a + a = 0111 + 0111 = 1110$, generate an **overflow** as the result should be positive

Examples

- $N = 8$, $a = 127 = 0111\,1111$, $b = 1_{10} = 0000\,0001$

$$a + b = 1000\,0000 = -128$$

overflow sign bit different from that of operands

- $N = 8$, $a = 125 = 0111\,1101$, $b = 2_{10} = 0000\,0010$

$$a + b = 0111\,1111$$

OK sign bit equal to that of operands

- $N = 6$, $a = -25 = 10\,0111$, $b = -13 = 11\,0011$

$$(-a) + (-b) = 00\,1010$$

overflow sign bit different from that of operands

- $N = 6$, $a = 25 = 01\,1001$, $b = -13 = 11\,0011$

$$a - b = 00\,1100$$

OK even a carry over the last column is generated since operands have different sign values.

Two's Complements Notation

Using the two's complement notation

- zero has only one representation
- the operation of sum is equal to that of natural numbers
- sums and subs may be realized using only one circuit computing

$$a - b = a + b_{2C}$$

  ad the sign bit is computed as any other bit.
- using N bits the range is $[\ -2^{N-1}\ \ldots\ (2^{N-1}-1)\ ]$

The representation over $N + k$ bits can be extracted as sign extension of the representation over N bits.

Example: $N = 4$, $-7 = 1001$ using $N = 8$ bits $-7 = 1111\,1001$.

Representing Integer Numbers using Excess Notation

- numbers are represented over N bits and the coding is called **excess** 2^{N-1}, example in picture we a coding excess 8
- we represent numbers $z \in [-2^{N-1} \dots 2^{N-1} - 1]$
- we code over N bits the binary value $2^{N-1} + z$ where 2^{N-1} is called **bias**
- the coding is similar to the two's complement with the sign bit reversed (1 for positive and 0 for negative)
- addition operations needs to be adjusted subtracting the bias
- this notation is not used for integer numbers but used in the representation of real numbers.

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	−1
0110	−2
0101	−3
0100	−4
0011	−5
0010	−6
0001	−7
0000	−8

Storing Real Numbers

Representation of Real Numbers

- representation of real numbers (rationals and irrationals) like

$$\frac{1}{3} \quad \frac{2}{7} \quad \ldots \pi \quad \sqrt{2} \quad \sqrt{3} \quad \ldots$$

  needs an **infinite** number of bits

- CPUs represent numbers using a **finite** number of bits

- for this reason we may only implement an approximate representation of real numbers.

There two possible approaches

- **fixed point**
- **floating point**

Both divide the sequence of N bits into several fields each with a precise maning and format.

Fixed Point Representation

Using **fixed point** the N bit word is divided in two fields

- one stores the fractional part
- the latter stores the integer part

Example: $N = 32$, $i = 16$ bits the integer part and $f = 16$ bits for the fractional

For example, to represent the number 13.25

- 13 in binary is 1101: $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$
- 0.25 in binary is 0.01: $0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1/4 = 0.25$

then, the representation in fixed point of 13.25 is

$$\underbrace{0000\,0000\,0000\,1101}_{\text{16 bit parte intera}} \;.\; \underbrace{0100\,0000\,0000\,0000}_{\text{16 bit parte frazionaria}}$$

We are assuming now to represent only positive numbers; to represent negative numbers we may use the MSB as sign bit.

Fixed Point Representation

Example: write the fixed point representation of number 22.875 using a word di 12 bits with 8 bits for the integer part and 4 bits for the fractional part.

- write 22 in binary

$$22 = 2^4 + 2^2 + 2^1$$

$$22 = 1\,0110$$

- write 0.875 in binary

$$0.875 = 0.5 + 0.25 + 0.125 = 2^{-1} + 2^{-2} + 2^{-3}$$

$$0.875 = 0.111$$

- the representation in fixed point is

$$0001\,0110\,.\,1110$$

Fixed Point Representation

Example: let $1000\,0001.0011$ the fixed point representation of a number with 12 bits, 8 for the fractional and 4 for the integer. The corrisponding value base 10 is:

- compute the base 10 value of integer part

$$1 \cdot 2^7 + 0 \cdot 2^6 + \ldots 0 \cdot 2^2 + 1 \cdot 2^0 = 128 + 1 = 129$$

- compute the base 10 value of fractional part

$$0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 1/8 + 1/16 = 0.1875$$

- the base 10 value of $1000\,0001.0011$ is 129.1875

Fixed Point Representation

Example: write the fixed point representation with $i = 8$ e $f = 8$ of number 23.625.

- write 23 as sum of positive powers 2

$$23 = 2^4 + 2^2 + 2^1 + 2^0$$

the binary representation is $1\,0111$

- write 0.625 as sum of negative powers 2

$$0.625 = 0.5 + 0.125 = 2^{-1} + 2^{-3}$$

getting the binary representation 0.101

- write the integer part using 8 bits and adding zeros to the left, and the fractional part on 8 bits adding zeros to the right

$$0001\,0111.1010\,0000$$

Fixed Point Representation

Esempio: write the smallest number that we can represent in fixed point using $i = 8$ and $f = 8$.

- the smallest number is
$$0000\,0000\ .\ 0000\,0001$$
- the integr part has value 0
- the fractional part has value $2^{-8} = 0.00390625$
- the corresponding value base 10 of $0000\,0000.0000\,0001$ is 0.00390625.

Fixed Point Representation

Esempio: write the largest number that we can represent in fixed point using $i = 8$ and $f = 8$.

- the largest number that we can represent is

$$1111\,1111.1111\,1111$$

- the base 10 value of integer part is

$$\sum_{h=0}^{7} 2^h = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

- the base 10 value of fractional part is

$$\sum_{h=1}^{8} 2^{-h} = 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128 + 1/256 = 0.99609375$$

- the base 10 value of $1111\,1111.1111\,1111$ is 255.99609375.

Fixed Point Representation

The **granularity** is the smallest difference between two consecutive numbers; this is directly related to the level of precision of the notation.

If the fractional part is represented using f bits, the **granularity** is 2^{-f}.

Example: assume $N = 8$, $i = 4$, $f = 4$

- let consider the number $r_0 = 0000.0000 = 0_{10}$
- the smallest number greater than r_0 is $r_1 = 0000.0001 = 1/16 = 0.0625_{10}$
- the difference $\delta = r_1 - r_0 = 0.0625 = 2^{-4}$ is the gap between the two numbers
- all numbers between r_0 e r_1 can **NOT** be represented !

In this exaple the granularity is $2^{-4} = 0.0625$.

Fixed Point Representation

If the fractional part can not be represent using f bits the value should be **truncated**.

Example: write the fixed point represention of the number $2 + \frac{5}{8}$ using $f = 2$ bits

- the binary representation of $2 + \frac{5}{8} = 2.625$ is

$$2 + \frac{4+1}{8} = 2 + \left(\frac{1}{2} + \frac{1}{8}\right) = 10\,.\,101$$

- fitting the representation into fixed point format with $f = 2$, the rightmost 1 should be removed getting

$$10\,.\,10 = 2.5$$

that is an approximation of the original value !

Fixed Point Representation

The fixed point representation is NOT used as it does not allow to use efficiently the bits when big and very small numbers are used.

Esempio: $N = 32$, $i = 16$, $f = 16$

- the fixed point representation of 60000.00 is

$$1110101001100000\ .\ 0000000000000000$$

the bits in the fractional part are all set to zero.

- the fixed point representation of 2^{-15} is

$$0000000000000000\ .\ 0000000000000010$$

in this case the bits of the integer part are all set to zero.

In the fixed point notation the number of bits for the integer and fractional part is fixed once for all and can not be changed accrding to the value of the number.

Floating Point Representation

The basic idea of **floating point** notation is to use the N bits word in efficient way, in particular to use the bits:

- to represents the fractional part when small numbers are used
- to represent the integer part when large numbers are used

allowing to represent small numbers with a smaller **granularity**.

To this end, we use the **normalized scientific** notation where each real number can be written in the format

$$\pm a.m \cdot b^e$$

with $1 \leq a < b$, m being the **mantissa** or **significand**, b the **base** and e the **exponent**.

Using the **normalized scientifc** notation the first digit before the radix point is always ≥ 1.

Floating Point Representation

Normalized Scientic Representation

$$x = \pm a.m \cdot b^e$$

Example in base $b = 10$:

- $347.65 = 3.4765 \cdot 10^2$, $a = 3$, $m = 4765$, $e = 2$
- $0.007653 = 7.653 \cdot 10^{-3}$, $a = 7$, $m = 653$, $e = -3$
- $310749 = 3.10749 \cdot 10^5$, $a = 3$, $m = 10749$, $e = 5$
- $0.00018 = 1.8 \cdot 10^{-4}$, $a = 1$, $m = 8$, $e = -4$

Floating Point Representation

Using he base $b = 2$, each binary number can be written in the format

$$\pm a.m \cdot b^e \quad 1 \leq a < 2$$

where m is the **mantissa**, $b = 2$ is the **base** and e the **exponent**.

Example:

- $110.1 = 1.101 \cdot 2^2 = 6.5_{10}$, $m = 101$, $e = 2$
- $0.001001 = 1.001 \cdot 2^{-3} = 0.140625_{10}$, $m = 001$, $e = -3$
- $1000.0 = 1.0 \cdot 2^3 = 8.0_{10}$, $m = 0$, $e = 3$

Using the base 2 each binary number has $a = 1$ then it can be omitted in the representation !

Floating Point Representation

IEEE 754 is the standard used to define **floating point** numbers.

For the **single precision** representation we use a $N = 32$ bits word divided into the fields:

- 1 sign bit s (the MSB)
- 8 bit for the exponent e represented in excess 127 notation $e = 127 + E$ where E is the exponent of the number representd in normalized scientific notation
- 23 bit for the matissa m

Example: represent the number 0.15625

It is not necessary to represent a since it is assumed to be always 1; it is named the hidden bit.

Floating Point Representation

For **single precision** numbers the IEEE 754 defines

- the mantissa m is represented in binary format using 23 bits
- the exponent e
  - is represents in binary format using 8 bits and the **excess k=127** notation

$$e = (E + 127)_2$$

where E is the exponent of the number represented using the scientific normal notation.

  - using 8 bits for the exponent we may represent 256 different (exponent) values
  - the negative exponents of numbers < 1, that is the values $E = [-127 \ldots -1]$ are mapped as $e = [0 \ldots 126]$
  - the positive exponents of numbers > 1, that is the values $E = [0 \ldots 128]$ are mapped onto $e = [127 \ldots 255]$
  - the exponent e is than alway ≥ 0 and we do not need to represent the sign.

Floating Point Representation

Example: write the IEEE 754 floating point representation of number -3.5.

The binary representaion of 3.5 is 11.1, and the correspondin normal scientific representation is

$$1.11 \cdot 2^1 = ((1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}) \cdot 2^1)_{10} = (2 + 1 + 0.5)_{10} = 3.5_{10}$$

Translating this into IEEE 754 we get

- $s = 1$
- $m = 1100\,0000\,0000\,0000\,0000\,000$, the mantissa should be represented on 23-bits adding zeros to the right if necessary
- $e = (127_{10} + 1_2) = 0111\,1111_2 + 1_2 = 1000\,0000_2$

The IEEE 754 format is then

$$\underbrace{1}_{\text{s}}\ \underbrace{10000000}_{\text{e = 8 bit}}\ \underbrace{11000000000000000000000}_{\text{m = 23 bit}}$$

The 1 preceding the decimal point (a) is not represented (hidden bit).

Floating Point Representation

In the IEEE 754 standard specific value of exponent are reserved

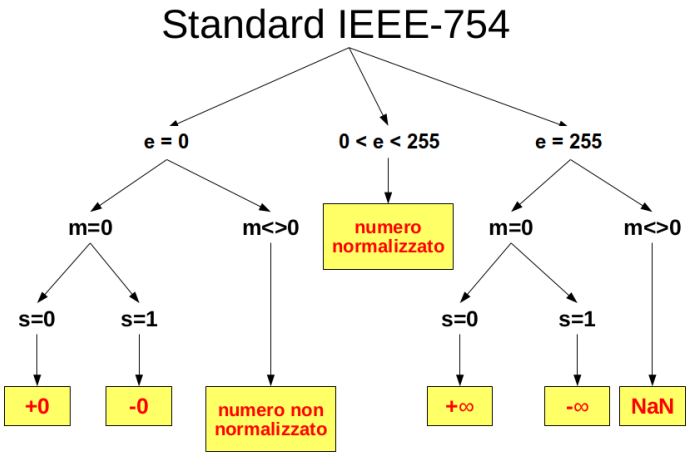
- $e = 0000\,0000 = 0_{10}$, corresponding to $E = -127$ is used to represent the zero if $m = 0$, otherwise the **denormalized** numbers if $m \neq 0$
- the values $1 \leq (e)_{10} \leq 254$ corresponding to $-126 \leq E \leq 127$ are used for **normalized** numbers
- $e = 1111\,1111 = 255_{10}$, corresponding to $E = 128$, is used to represent the **infinite** if $m \neq 0$, and **not a number NaN** if $m = 0$.

denormalized or **subnormalized** are those numbers between zero and the smallest normalized number that we can represent. These a very small numbers that can not be represented using the normalized format.

NaN is used for numbers that can not be representd, es. $\sqrt{-2}$.

infinite is used for very large numbers that can not be represented.

Floating Point Representation



The standard IEEE 754 use two different representations for zero: $+0$ e -0.

Excercise

Es1: Extract the base 10 value of the exponent of a single-precision floating number with $e = 0011\,1011$:

$$E_{10} = (0011\,1011)_{10} - 127 = 2^5 + 2^4 + 2^3 + 2^1 + 2^0 - 127 = 32 + 16 + 8 + 2 + 1 - 127 = 59 - 127 = -68$$

Es2: Extract the base 10 value of a single-precision number representd by

$$0\ 1000\,0000\ 100\,0000\,0000\,0000\,0000\,0000$$

- the sign bit is positive
- the value of exponent is $E_{10} = (10000000)_{10} - 127 = 2^7 - 127 = 128 - 127 = 1$
- the value of mantissa is $m_{10} = (1.10000000000000000000000)_{10} = 2^0 + 2^{-1} = 1.5$

the base 10 value is $1.5 \cdot 2^1 = 3.0$.

Excercise

Write the 32-bit IEEE 754 representation of number -4.5.

- sign bit is $s = -1$
- binary representation of $4.5_{10} = (4 + 0.5)_{10} = (2^2 + 2^{-1})_{10} = (100.1)_2$
- the normalized form of 4.5 è $1.001 \cdot 2^2$ (we moved the radic point by two positions)
- the mantissa is $m = 00100000000000000000000$
- the exponent is $e = 2 + 127 = 129 = (1000\,0010)_2$

The 32-bit IEEE 754 representation of number -4.5 is

$$1\ 1000\,0010\ 00100000000000000000000$$

Floating Point Representation

Practical Exercise:

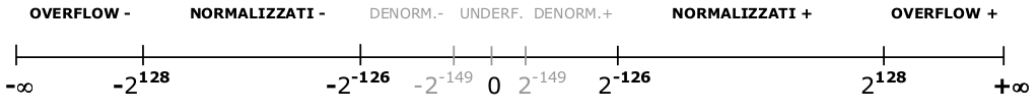
We can use the *float-rep.c* example to print the bit representation of a floating-point number:

```
gcc float-rep.c -o float-rep
```

Now, run the executable *./float-rep* and enter a number, its single-precision floating-point representation (as stored in memory) will be shown to you, next to the number it actually represent.

You can play trying several numbers...

Floating Point Representation



- for normalized numbers we have $-126 \leq E \leq 127$
- the absolute value of the largest normalized number is

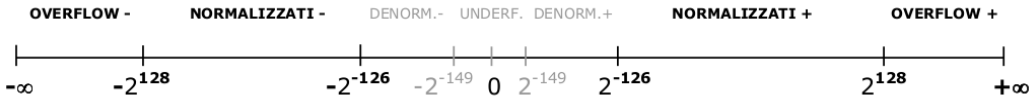
$$1.\underbrace{11\ldots111}_{23\ bit} \cdot 2^{127} \approx (1+1) \cdot 2^{127} = 2 \cdot 2^{127} = 2^{128} \approx 10^{38}$$

  we can represent number in the range $-2^{128} < x < 2^{128}$ (but not all numbers !)
- ranges $(-\infty, -2^{128})$ and $(2^{128}, +\infty)$ can not be represented (overflow)
- the absolute value of the smallest normalized number is

$$1.\underbrace{00\ldots000}_{23\ bit} \cdot 2^{-126} = 2^{-126} \approx 10^{-38}$$

- $(-2^{-126}, 0)$, $(0, 2^{-126})$ are the **denormalized** ranges including **denormal** numbers

Floating Point Representation: Denormalized



The **denormalized** range $(-2^{-126}, 0)$, $(0, 2^{-126})$ includes **subnormals** and **underflow** numbers.

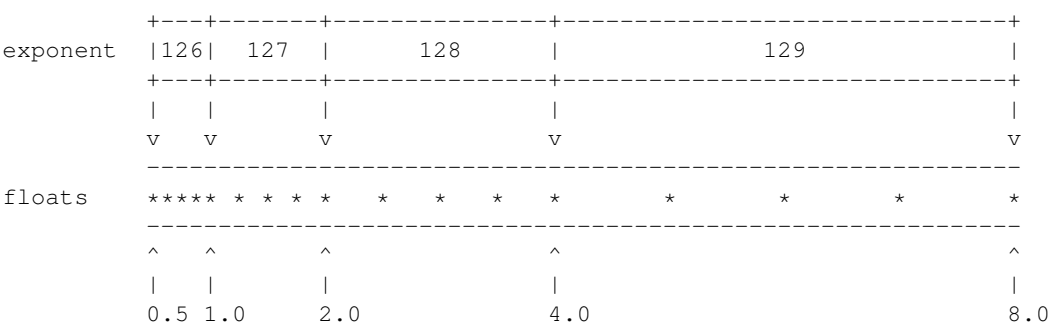
- the **subnormals** are numbers that can not be normalizedd since the exponent does not fits into the available bits
- in this cases the hidden bit is equal to zero and then the represention is not in the normal format e.g.:

$$2^{-127} = (0.1)_2 \times 2^{-126} \qquad 2^{-128} = (0.01) \times 2^{-126}$$

- the smallest positive subnormal number is $2^{-23} \times 2^{-126} = 2^{149}$
- the largest negative subnormal number is $-2^{-23} \times 2^{-126} = -2^{149}$
- **subnormal** numbers are stored with biased exponent zero but are decoded with the value of the smallest allowed exponent for normals -126
- production of a subnormal numbers is sometimes called **gradual underflow** because it allows a computation to lose precision slowly when the result is small.

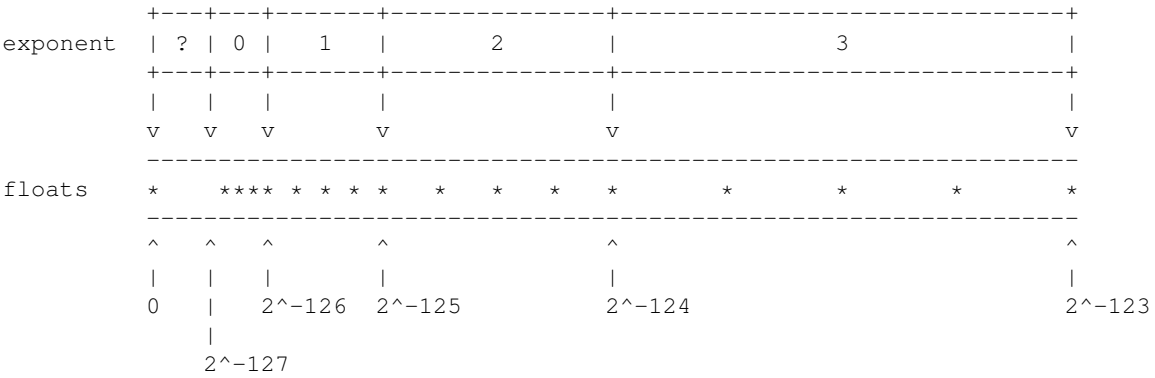
Floating Point Representation

We can try to have a geometric intuition about what we have learn: if we plot IEEE 754 floating point numbers on a line for each given exponent, it looks something like this:



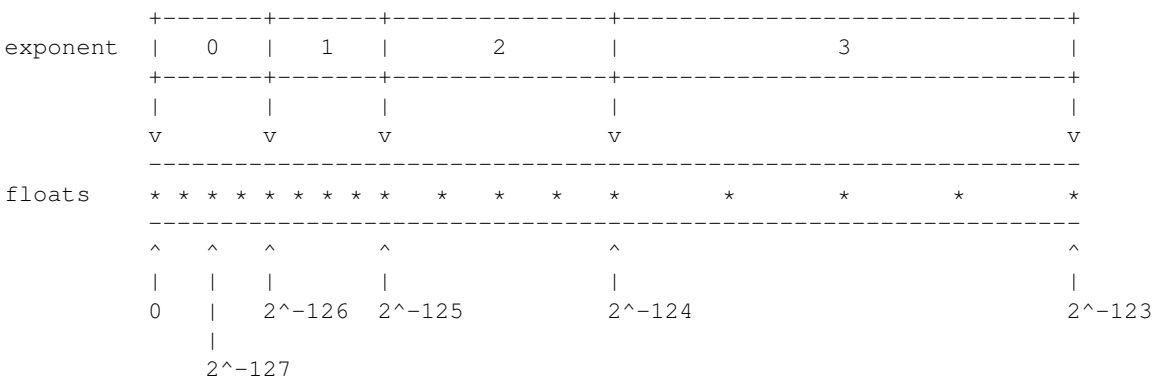
Floating Point Representation: Denormalized

To understand why the further complexity of subnormals was introduced, this is how it would hypothetically look like without:



Floating Point Representation: Denormalized

...while with subnormals, it looks like this:



Floating Point Representation

- the mantissa m can represent 2^{23} different values
- the exponent e can represent 254 different values (0 and 255 are not used for normalized numbers)
- the sign bit can be either 0 or 1
- the zero is represented using $e = 0$

in total then we can represnt

$$2 \times 254 \times 2^{23} + 1 = 4\,261\,412\,865$$

numbers compared to a set dense, infinite and not limited of real numbers !

The set of numbers represented by the IEEE 754 format is **NOT** dense and it is is limited !

Arithmetic of computers is an approximation of real one.

Floating Point Representation

- the ranges of positive and negative numbers are not contigous there are **gaps**
- within each range numbers are not uniformely ditributed, and the distance between big numbers is larger than distance between small numbers, focusing the precision where it is more necessary.

Example

- $x_1 = 1\,.\,00000000000000000000000 \cdot 2^{100} = 2^{100}_{10}$ e

  $x_2 = 1\,.\,00000000000000000000001 \cdot 2^{100} = 2^{100}_{10} + 2^{100-23}_{10} = 2^{100}_{10} + 2^{77}_{10}$

  the distance is: $\delta = x_2 - x_1 = 2^{100}_{10} + 2^{77}_{10} - 2^{100}_{10} = 2^{77} \approx 10^{23}$

- $x_1 = 1\,.\,00000000000000000000000 \cdot 2^{0} = 2^{0}_{10} = 1_{10}$ e

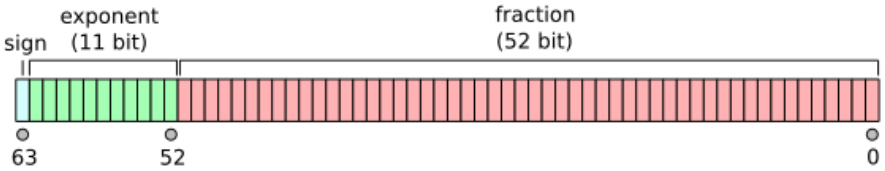
  $x_2 = 1\,.\,00000000000000000000001 \cdot 2^{0} = 2^{0}_{10} + 2^{-23}_{10}$

  the distance is: $\delta = x_2 - x_1 = 2^{0}_{10} + 2^{-23}_{10} - 2^{0}_{10} = 2^{-23} \approx 10^{-7}$

Base 10 - IEEE 754 converter: `https://www.h-schmidt.net/FloatConverter/IEEE754.html`

Floating Point Representation

The IEEE 754 standard defines also **double precision** format utilizzando a $N = 64$ bit word divided as following

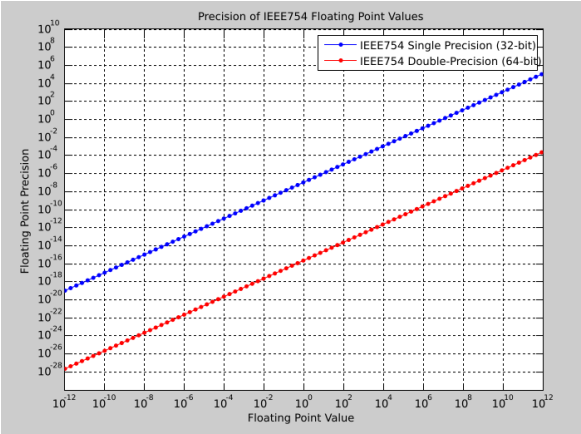


- 1 sign bit
- 11 bits for the esponente
- 52 bits for the mantissa

It is also defined the standard for $N = 128$ bits that allows to have a preciosn four times better w.r.t. the single precision. IN this case we have the sign bit, 15 bits of exponent and 112 bits for the mantissa.

Floating Point Representation

The figure below shows the absolute precision for both single- and double-precision formats over a range of values



Floating Point Representation

Practical Exercise:

We can use the *float-distance.c* example to print the the number of representable double precision floating-point values in an interval:

```
g++ float-distance.cpp -o float-distance
```

Now, run the executable *./float-distance*

- Is the number of representable values the same for intervals of the same “size”?
- Modify the code and try to let it show you other intervals; some of the interesting ones are commented in the source code.

You can play trying several ranges...

Floating Point Representation

Gerald Jay Sussman, Professor at MIT, once said:

"Nothing brings fear to my heart more than a floating point number."

Practical Exercise:

We can use the *commutative-sum.c* example to understand why we should have the same fear:

```
g++ commutative-sum.cpp -o commutative-sum
```

This code is very simple, it just:

- fills one array with a random uniform distribution of floating-point values;
- copies this array in a second one;
- sorts the values in the second array (but not in the first array);
- sums togheter all the values of the first array;
- sums togheter all the values of the second array;

Floating Point Representation

Practical Exercise:

You can compile the code and run it, then try to answer to these questions:

- Are the two results equal?
- Is then the sum operation commutative?
- How do you justify this?
- Ok, at the end the error is not “so large”, why to worry?
- What if the random values are in an interval larger than [-1,1]?
- Modify the code and try larger intervals.
- Do you feel the fear in your heart?

Floating Point Representation

If you are not scared enough try also this:

Practical Exercise:

Have a look to the source code of the example *my-math.c* and then compile it with different flags:

```
gcc my-math.c -o my-math
gcc -O3 my-math.c -o my-optimized-math
gcc -Ofast my-math.c -o my-very-optimized-math
```

- Running the different executables, are the results the same?
- What have you done setting -Ofast?
- Try: “man gcc” to find it out
- Which one of the different flags activated by -Ofast is changing the result?
- Try to activate each of them to find it out:
  `gcc -O3 -feachflag my-math.c -o my-custom-optimized-math`

Further references

- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, March, 1991 issue of Computing Surveys.
- John Farrier, *Demystifying Floating Point* CppCon 2015. https://www.youtube.com/watch?v=k12BJGSc2Nc
- *An Interview with the Old Man of Floating-Point*, Reminiscences elicited from William Kahan by Charles Severance. https://people.eecs.berkeley.edu/%7Ewkahan/ieee754status/754story.html