ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 2 (II)
Floating point

Computer Structure

Bachelor in Computer Science and Engineering

Contents

More representation necessities…

- How to represent?
  - Very large numbers: $30.556.926.000_{(10}$
  - Very small numbers: $0.0000000000529177_{(10}$
  - Fractional numbers: 1.58567

Reminder
Example of failure…

- **Ariane 5** explosion (first flight)
  - Sent by ESA in June 1996
  - Cost of development:
    10 years and 7 billion dollars
  - Exploded 40 seconds after launch, at 3700 meters altitude.
  - Failure due to total loss of altitude information:
    - The inertial reference system software performed the conversion of a 64-bit floating point real value to a 16-bit integer value.
    - The number to be stored was greater than 32767 (the largest 16-bit signed integer) and a conversion failure and exception occurred.

Fixed point [racionals]

- The position of the binary point is fixed and the weights associated with the decimal places are used.

- Example:

$$1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$$

Fractional values in binary with fixed point

- Example with 6 bits:

$$\text{xx.yyyy}$$

$$2^{1} \quad 2^{0} \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4}$$

- Example:
  $10{,}1010_{(2} = 1 \times 2^{1} + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$

- Using this fixed point, the range is:
  - [0 a 3.9375 (almost 4)]

Félix García-Carballeira, Alejandro Calderón Mateos

Fractional powers of 2

i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.251/4	
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	

Contents

Félix García-Carballeira, Alejandro Calderón Mateos

Floating-point numbers

$$\underbrace{9.12}_{\text{mantissa}}\,{}_{\underbrace{10}_{\text{radix (base)}}} \times 10^{\underbrace{25}_{\text{exponent}}}$$

- Each number has a mantissa and an $^{\text{exponent}}$
- Scientific notation (in decimal): normalized form
  - Only one digit different to 0 on the left of decimal point
- The number is adapted to the order of magnitude of the value to be represented, by translating the *decimal point* by using the exponent

Scientific notation in binary

mantissa

exponent

$$1.0_{(2} \text{ x } 2^{-1}$$

binary point

radix (base)

- Normalized form:
  One 1 (only one digit) in the left of the binary point
  - Normalized: $1.0001 \text{ x } 2^{-9}$,
  - Not normalized: $0.0011 \text{ x } 2^{-8}$, $10.0 \text{ x } 2^{-10}$

IEEE 754 Floating Point Standard [rationals]

1 0010...0101 00010...001

sign **exponent** **mantissa**

- Floating point standard used in most computers.
- **Characteristics** (unless special cases):
  - Exponent: excess-k with bias $k = 2^{num\_bits\_in\_exponent - 1} - 1$
  - Mantissa: sign-magnitude, <u>normalized</u>, with <u>implicit bit</u>
- Different **formats**:
  - **Single precision**: 32 bits (sign: 1, exponent: 8, mantissa: 23 and bias: 127)
  - **Double precision**: 64 bits (sign: 1, exponent: 11, mantissa: 52 and bias: 1023)
  - **Quad-precision**: 128 bits (sign: 1, exponent: 15, mantissa: 112 and bias: 16383)

Normalization and implicit bit

- Normalization

  In order to normalize the mantissa, the exponent is adjusted to have a most significant bit of value 1

  - Example: 10010000000000000000000000 x 2^3 (already normalized)
  - Example: 000100000000010101 x 2^3 (is not)

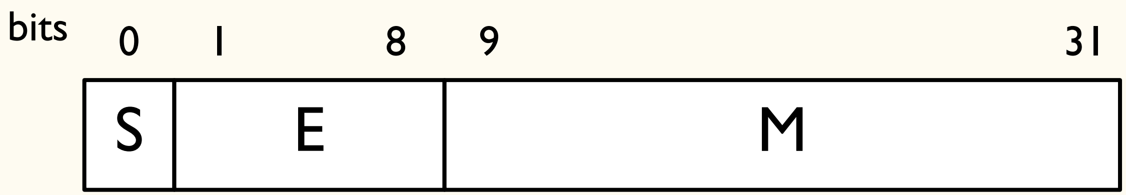
    100000000010101000 x 2^0 (now it is)

- Implicit bit

  Once normalized, since the most significant bit is 1, it is **not** stored to leave space for one more bit (increases accuracy).

  - This makes it possible to represent mantissa with one bit more

IEEE Standard 754 (single precision)



S: sign (1 bit)
E: exponent (8 bits)
M: mantissa (23 bits)

- The value is computed (unless special cases) as:

$$\mathbf{N = (-1)^S \times 2^{E-127} \times 1.M}$$

where:

S = 0 for positive numbers, S = 1 for negative numbers

0 < E < 255 (E=0 y E=255 are special cases)

$00000000000000000000000 \leq M \leq 11111111111111111111111$

IEEE Standard 754 (single precision) [rationals]

- Special cases:

$(-1)^s \times 0.\text{mantissa} \times 2^{-126}$

Exponent	Mantissa	Special value
0 (0000 0000)	0	+/- 0 (depends on sign)
0 (0000 0000)	≠ 0	Number NOT normalized
255 (1111 1111)	≠ 0	NaN (0/0, sqrt(-4), ….)
255 (1111 1111)	0	+/- infinite (depends on sign)
1-254	Any	Normalized number (no special)

$(-1)^s \times 1.\text{mantissa} \times 2^{\text{exponent}-127}$

Examples

S	E	M	N
1	00000000	00000000000000000000000	-0 (Exception 0) E=0 y M=0.
1	01111111	00000000000000000000000	$-2^0 \times 1.0_2 = -1$
0	10000001	11100000000000000000000	$+2^2 \times 1.111_2 = +2^2 \times (2^0+2^{-1}+2^{-2}+2^{-3}) = +7.5$
0	11111111	00000000000000000000000	∞ (Exception ∞) E=255 y M=0
0	11111111	10000000000000000000001	NaN (Not a Number) E=255 y $M \neq 0$.

Example

a) Calculate the value in decimal associated to this number
0 10000011 11000000000000000000000
represented in IEEE 754 single precision

Example (solution)

a) Calculate the value in decimal associated to this number
0 10000011 11000000000000000000000
represented in IEEE 754 single precision

> a) Sign bit: 0 $\Rightarrow (-1)^0 = +1$
> b) Exponent: $10000011_2 = 131_{10} \Rightarrow E - 127 = 131 - 127 = 4$
> c) Mantissa: 11000000000000000000000 $\Rightarrow 1 \times 2^{-1} + 1 \times 2^{-2} = 0.75$
>
> The decimal value is $+1 \times 2^4 \times 1.75 = +28$

Exercise

b) Represent the number -9 using IEEE 754 single precision

Exercise (Solution)

b) Represent the number -9 using IEEE 754 single precision

$-9_{10} = -1001_2 = -1001_2 \times 2^0 = -1.001_2 \times 2^3$ (normalized mantissa)

a) Sign: negative ⇨ S=1
b) Exponent: 3+127 (bias) = 130 ⇨ 10000010
c) Mantissa: 1,001 (impl. bit) ⇨ 00100000000000000000000

-9 is represented by 1 10000010 00100000000000000000000

IEEE Standard 754 (single precision) [rationals]

- Range of representable magnitudes (regardless of sign):
  - Smallest normalized:
    $2^{1-127} \times 1.00000000000000000000000_2$
  - Largest normalized:
    $2^{254-127} \times 1.11111111111111111111111_2$
  - Smallest not normalized :
    $2^{-126} \times 0.00000000000000000000001_2$
  - Largest not normalized :
    $2^{-126} \times 0.11111111111111111111111_2$

$(-1)^s * 0.\text{mantisa} * 2^{-126}$

Exponent	Mantissa	Special value
0	$\neq 0$	Not normalized
1-254	any	Normalized

$(-1)^s * 1.\text{mantisa} * 2^{\text{exponente}-127}$

Félix García-Carballeira, Alejandro Calderón Mateos

IEEE Standard 754 (single precision) [rationals]

- Range of representable magnitudes (regardless of sign):
  - Smallest normalized:
    $2^{1-127} \times 1.00000000000000000000000_2 = 2^{-126}$
  - Largest normalized:
    $2^{254-127} \times 1.11111111111111111111111_2 = 2^{127} \times (2 - 2^{-23}) = 2^{128} \times (1 - 2^{-24})$
  - Smallest not normalized :
    $2^{-126} \times 0.00000000000000000000001_2 = 2^{-149}$
  - Largest not normalized :
    $2^{-126} \times 0.11111111111111111111111_2 = 2^{-126} \times (1 - 2^{-23})$

Tip:

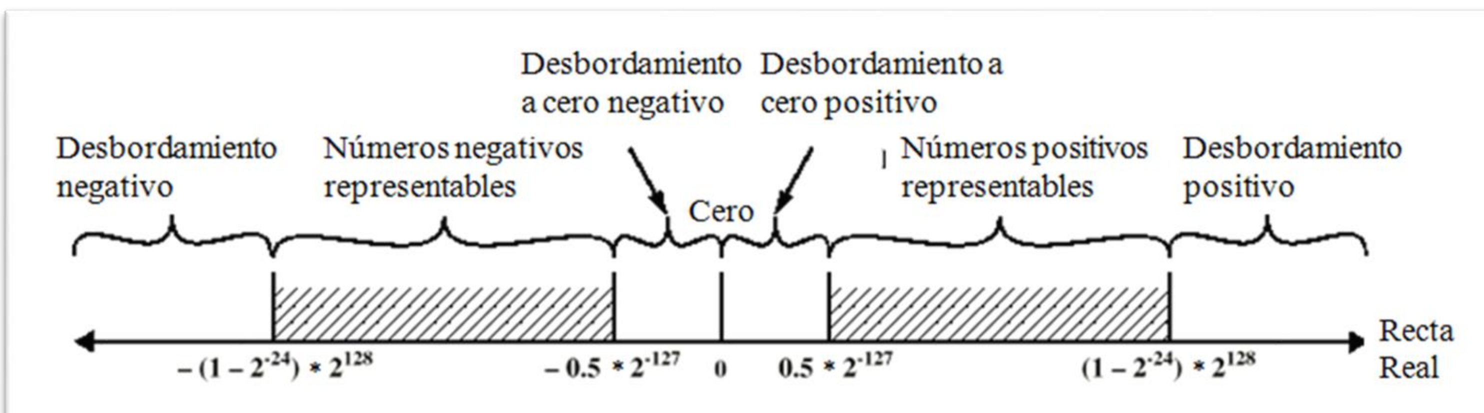
$$
\begin{array}{rll}
 & 1.11111111111111111111111_2 & = X \\
+ & 0.00000000000000000000001_2 & = 2^{-23} \\
\hline
 & 10.00000000000000000000000_2 & = 2
\end{array}
$$

$$X = 2 - 2^{-23}$$

Félix García-Carballeira, Alejandro Calderón Mateos

IEEE Standard 754 (single precision) [rationals]

- Range of representable magnitudes (regardless of sign):
  - Smallest normalized:
    $2^{1-127} \times 1.00000000000000000000000_2 = 2^{-126} = 2^{-127} \times 0.5$
  - Largest normalized:
    $2^{254-127} \times 1.11111111111111111111111_2 = 2^{127} \times (2 - 2^{-23}) = 2^{128} \times (1 - 2^{-24})$
  - Smallest not normalized :
    $2^{-126} \times 0.00000000000000000000001_2 = 2^{-149}$
  - Largest not normalized :
    $2^{-126} \times 0.11111111111111111111111_2 = 2^{-126} \times (1 - 2^{-23})$



Exercise

- How many *floats* (single precision floating point numbers) are between 1 and 2 (not included)?

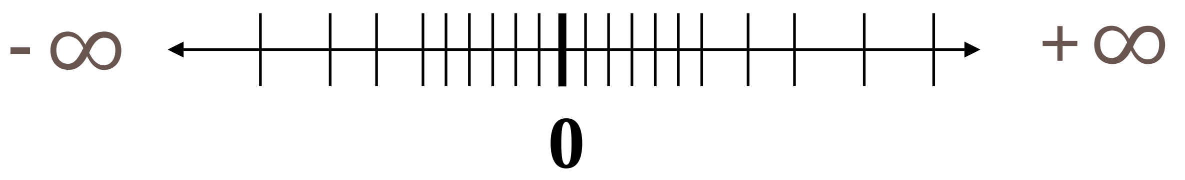
- How many *float* (single precision floating point numbers) are between 2 and 3 (not included)?

Exercise (Solution)

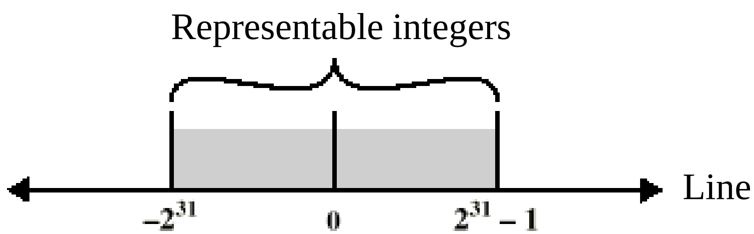
- How many *floats* (single precision floating point numbers) are between 1 and 2 (not included)?
  - $1 = 1.00000000000000000000000 \times 2^0$
  - $2 = 1.00000000000000000000000 \times 2^1$
  - Between 1 and 2 there are 2^{23} numbers
- How many *float* (single precision floating point numbers) are between 2 and 3 (not included)?
  - $2 = 1.00000000000000000000000 \times 2^1$
  - $3 = 1.10000000000000000000000 \times 2^1$
  - Between 2 and 3 there are 2^{22} numbers

Discrete representation

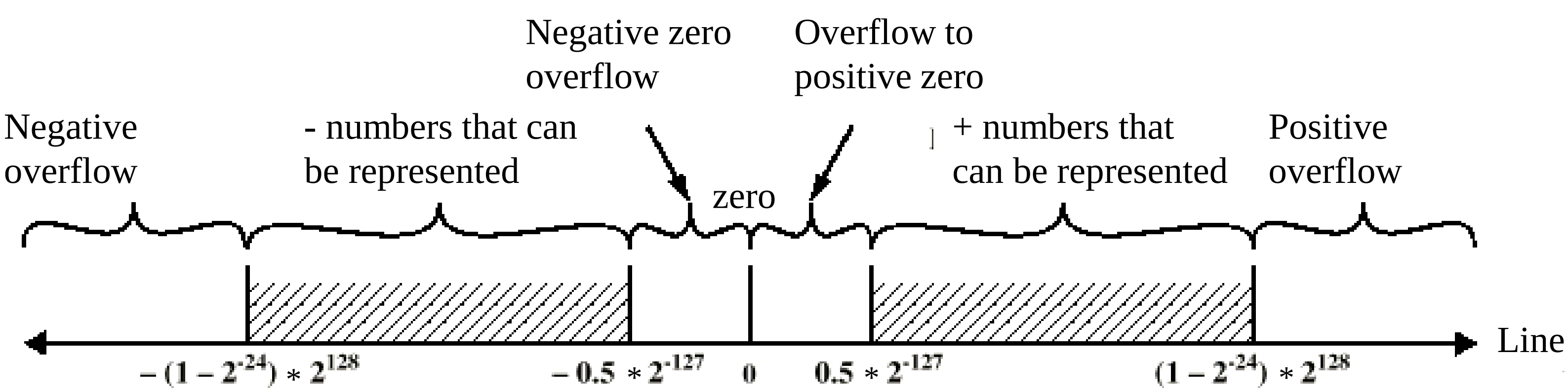
- Variable resolution:
  denser near zero, less towards infinity



Representable numbers



(a) Two's complement integers



(b) Floating point numbers

Example 1
inaccuracy

0.4 →

0	01111101	10011001100110011001101

3.9999998 e-1

0.1 →

0	01111011	10011001100110011001100

9.9999994 e-2

Example 2
inaccuracy

- How does C performs a division?

t2.c

```
#include <stdio.h>

int main ( )
{
   float a ;

   a = 3.0/7.0 ;
   if (a == 3.0/7.0)
          printf("Equal\n") ;
   else printf("Not equal\n") ;
   return (0) ;
}
```

Example 2
inaccuracy

- How does C performs a division?

t2.c

```
#include <stdio.h>

int main ( )
{
   float a ;

   a = 3.0/7.0 ;
   if (a == 3.0/7.0)
         printf("Equal\n") ;
   else printf("Not equal\n") ;
   return (0) ;
}
```

```
$ gcc -o t2 t2.c
$ ./t2
Not equal
```

Example 2
inaccuracy

- How does C performs a division?

t2.c

```
#include <stdio.h>

int main ( )
{
   float a ;

   a = 3.0/7.0 ;
   if (a == 3.0/7.0)
         printf("Equal\n") ;
   else printf("Not equal\n") ;
   return (0) ;
}
```

float

double

```
$ gcc -o t2 t2.c
$ ./t2
Not equal
```

Example 3
inaccuracy

- The associative property is not always satisfied

  ¿ a + (b + c) = (a + b) + c ?

t1.c

```
#include <stdio.h>

int main ( )
{
      float x, y, z ;

      x = 10e30;   y = -10e30;   z = 1;
      printf("(x+y)+z = %f\n",(x+y)+z) ;
      printf("x+(y+z) = %f\n",x+(y+z)) ;

      return (0) ;
}
```

Example 3
inaccuracy

- The associative property is not always satisfied

¿ a + (b + c) = (a + b) + c ?

t1.c

```
#include <stdio.h>

int main ( )
{
      float x, y, z ;

      x = 10e30;   y = -10e30;   z = 1;
      printf("(x+y)+z = %f\n",(x+y)+z) ;
      printf("x+(y+z) = %f\n",x+(y+z)) ;

      return (0) ;
}
```

```
$ gcc -o t1 t1.c
$ ./t1
(x+y)+z = 1.000000
x+(y+z) = 0.000000
```

Floating-point is not associative

- Floating-point is not associative
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$
  - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$
    $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = 0.0$
  - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
    $= (0.0) + 1.0 = 1.0$
- Floating point operations are not associatives
  - Results are approximated
  - 1.5×10^{38} is so much larger than 1.0
  - $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Example

int → float → int

```
if (i == (int)((float) i)) {
      printf("true");
}
```

- Not always prints `"true"`
- Most integer values (specially larger ones) don't have an exact floating point representation
- What about `double`?

Example

- The number 133000405 in binary is:
  - 111111011010110110011010101 (27 bits)
- $111111011010110110011010101 \times 2^0$
- When is normalized:
  - $1.11111011010110110011010101 \times 2^{26}$
  - S = 0 (positive)
  - $e = 26 \rightarrow \quad E = 26 + 127 = 153$
  - M = 11111011010110110011010 (last 3 bits are lost)
- The normalized number stored is:
  - $1.11111011010110110011010 \times 2^{26} =$
  - $111111011010110110011010 \times 2^3 = 133000400$

Example

float → int → float

```
if (f == (float)((int) f)) {
    printf("true");
}
```

- Not always true
- Numbers with decimals do not have integer representation

Rounding

- Rounding removes less significant digits from a number to obtain an approximate value.

- Types of rounding:
  - Round to + ∞
    - Round it "up": $2.001 \rightarrow 3$, $-2.001 \rightarrow -2$
  - Round to - ∞
    - Round it "down": $1.999 \rightarrow 1$, $-1.999 \rightarrow -2$
  - Truncate
    - Discard last bits: $1.299 \rightarrow 1.2$
  - Round to nearest (ties to even)
    - $2.4 \rightarrow 2$, $2.6 \rightarrow 3$, $-1.4 \rightarrow -1$
    - If number falls midway then it is rounded to the nearest value with an even least significant digit ($+23.5 \rightarrow +24 \leftarrow +24.5$; $-23.5 \rightarrow -24 \leftarrow -24.5$)

Rounding

- Rounding means losing accuracy.
- Rounding occurs:
  - When moving to a representation with fewer representables:
    - E.g.: A value from double to single precision
    - E.g.: A floating point value to integer
  - When performing arithmetic operations:
    - E.g.: After adding two floating-point numbers (using guard bits)

Guard bits

- Guard digits are used to improve accuracy:
  - FP hardware internally includes additional bits for operations
  - After operation, guard bits are eliminated: rounding

- Example: $2.65 \times 10^0 + 2.34 \times 10^2$

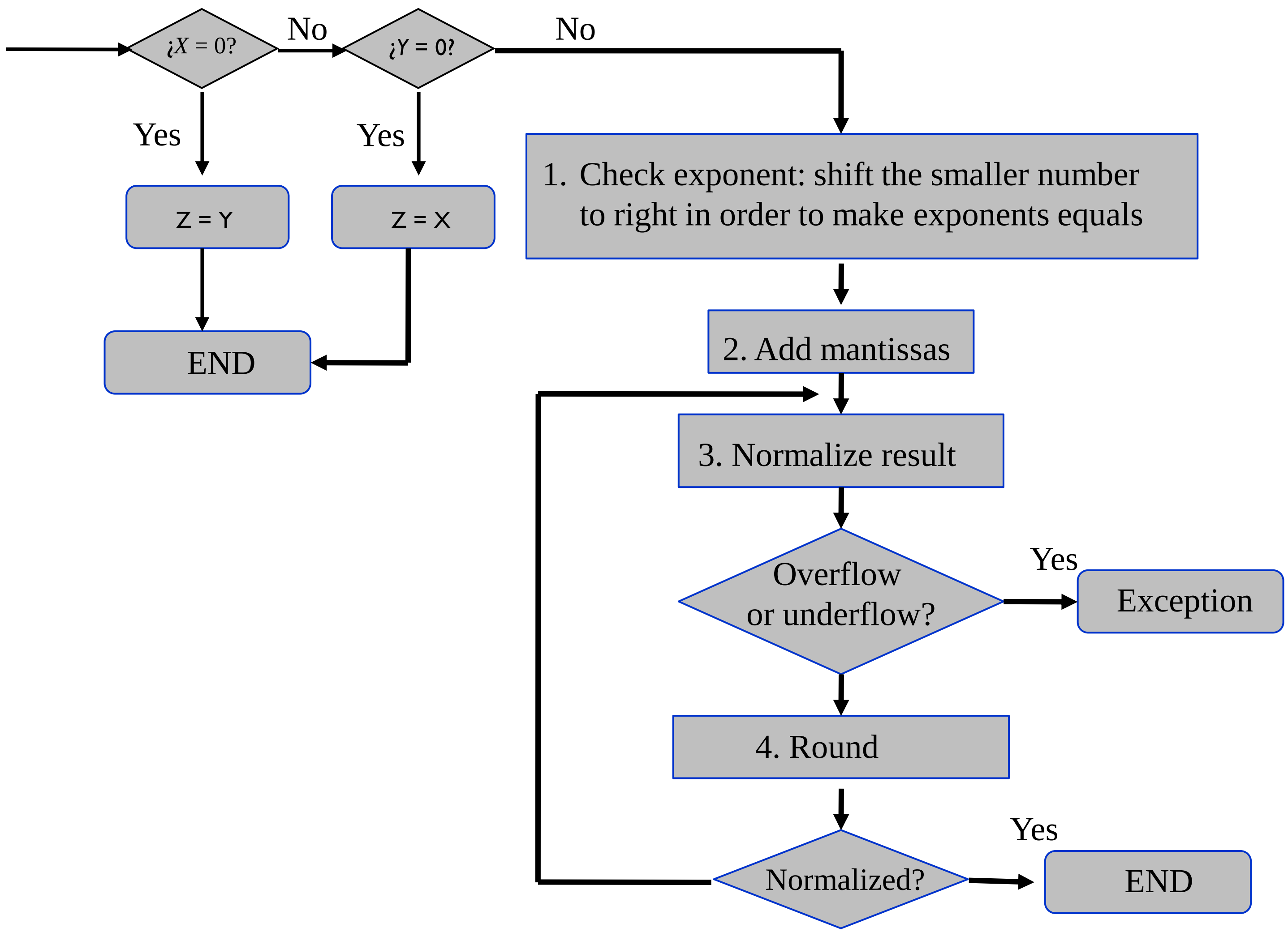
	WITHOUT guard bits	WITH guard bits
1.- equalize exponents	0.02×10^2 $+ 2.34 \times 10^2$	0.0265×10^2 $+ 2.3400 \times 10^2$
2.- add	2.36×10^2	2.3665×10^2
3.- round	2.36×10^2	2.37×10^2

Floating point operations

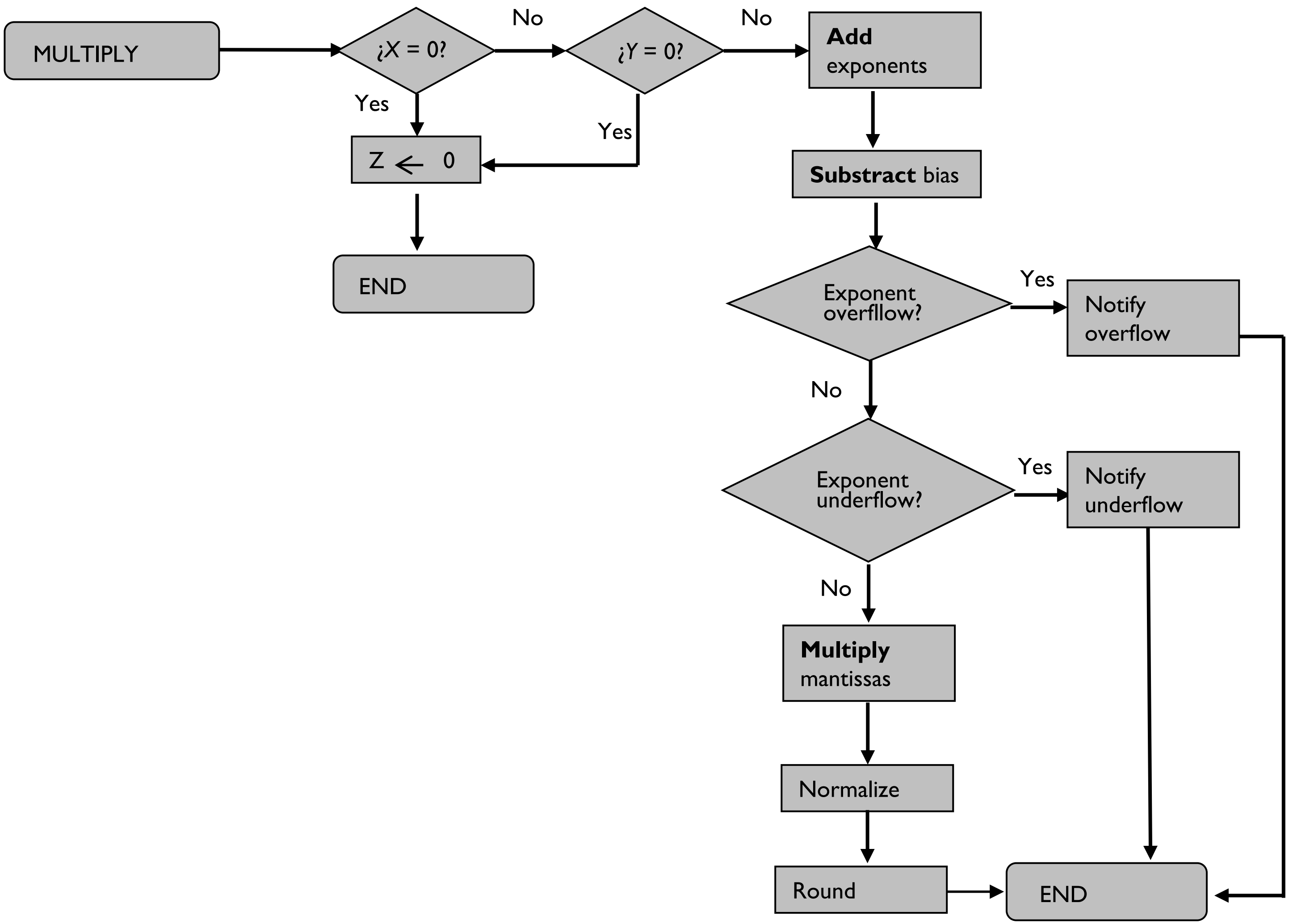
- Add
- Subtract
  1. Check zero values.
  2. Equalize exponents (shift smaller number to the right).
  3. Add/subtract mantissa.
  4. Normalize the result.

- Multiply
- Divide
  1. Check zero values.
  2. Add/subtract exponents.
  3. Multiply/divide mantissa (taking into account the sign).
  4. Normalize the result.
  5. Rounding the result.

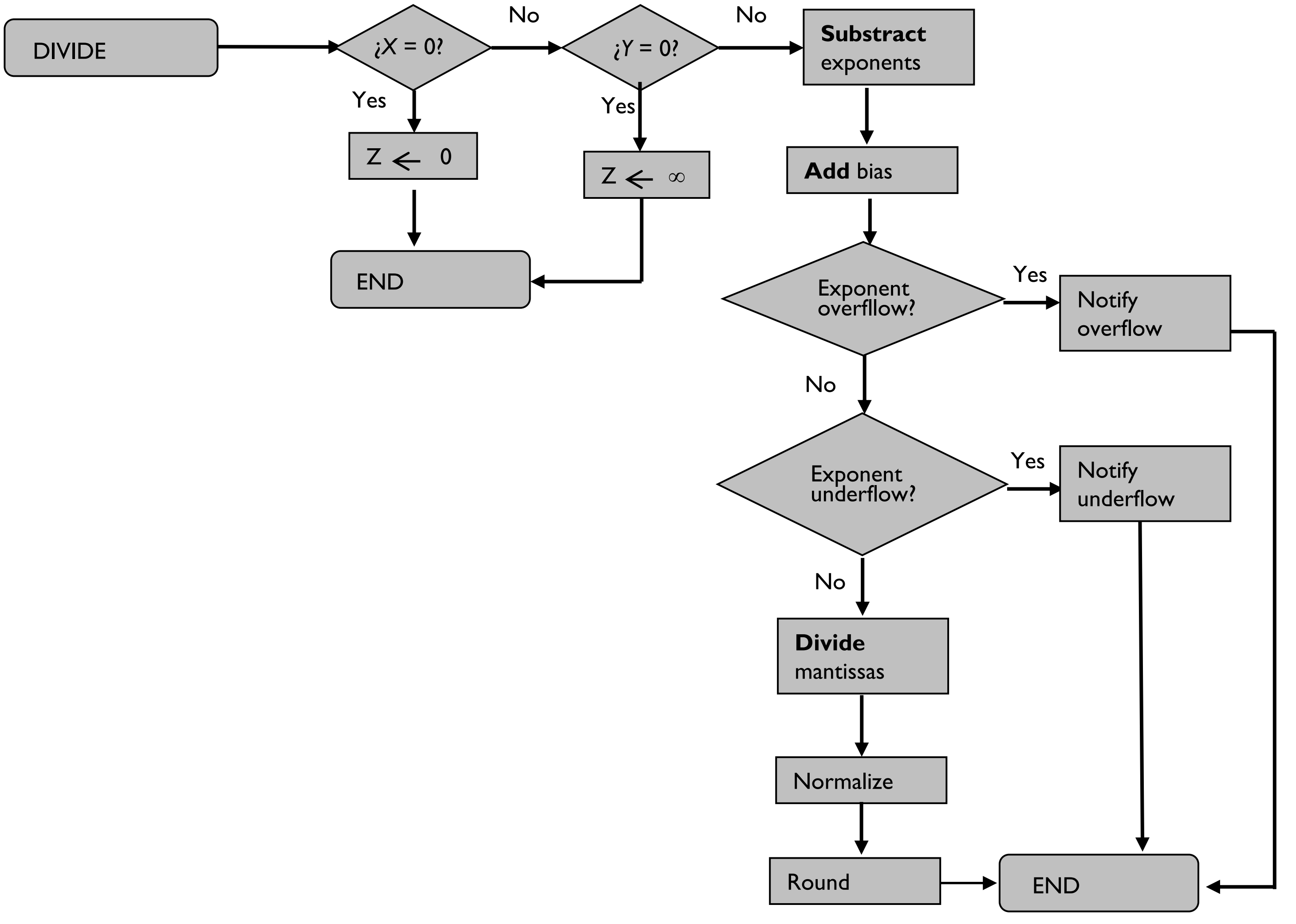
Additions and subtractions: Z=X+Y y Z=X-Y



Multiplication: Z=X*Y



Division: Z=X/Y



Exercise

- Using the IEEE 754 format, add 7.5 and 1.5 step by step.

Solution

1) $7.5 + 1.5 =$
2) $1.111*2^2 + 1.1*2^0 =$
3) $1.111*2^2 + 0.011*2^2 =$
4) $10.010*2^2 =$
5) $1.0010*2^3$

To binary

Equalize exponents

Add

Adjust exponents

Solution

- Representation of the numbers

7.5 → 0 10000001 11100000000000000000000

\+ **1.5** → 0 01111111 10000000000000000000000

Solution

- Splitting exponents and mantissas, and adding implicit bit

	7.5 →	0	10000001	1.11100000000000000000000
+	**1.5** →	0	01111111	1.10000000000000000000000

Solution

- Equalize exponents

7.5 →	0	10000001	1.11100000000000000000000
+ **1.5** →	0	01111111	1.10000000000000000000000

Solution

- Equalize exponents

	Sign	Exponent	Mantissa
7.5 →	0	10000001	1.11100000000000000000000
+ 1.5 →	0	01111111	1.10000000000000000000000

+1 1

Solution

- Equalize exponents

7.5 →	0	10000001	1.11100000000000000000000
		+1	1
+ **1.5** →	0	10000000	0.1100000000000000000000

Félix García-Carballeira, Alejandro Calderón Mateos

Solution

- Equalize exponents

	7.5 →	0	10000001	1.11100000000000000000000
			+1	1
+	1.5 →	0	10000001	0.01100000000000000000000

Solution

- Add mantissas

	Value	Sign	Exponent	Mantissa
	7.5 →	0	10000001	1.11100000000000000000000
+	**1.5** →	0	10000001	0.01100000000000000000000

Félix García-Carballeira, Alejandro Calderón Mateos

Solution

- Normalize result…

	7.5 →	0	10000001	1.11100000000000000000000
+	1.5 →	0	10000001	0.01100000000000000000000
	9 →	0	10000001	10.01000000000000000000000

There is carry,
non-normalized mantissa

Solution

- Normalize result…

	7.5 →	0	10000001	1.11100000000000000000000
+	1.5 →	0	10000001	0.01100000000000000000000
	9 →	0	10000001	10.01000000000000000000000

+1 ←→ 1

There is carry,
non-normalized mantissa

Solution

7.5 →	0	10000001	1.11100000000000000000000
+ 1.5 →	0	10000001	0.01100000000000000000000
9 →	0	10000010	1.00100000000000000000000

Solution

- Eliminate the implicit bit and store the result

9 → 0 10000010 00100000000000000000000

Exercise

- Using the IEEE 754 format, multiply 7.5 and 1.5 step by step.

Solution

- Representation of the numbers

7.5 → 0 10000001 11100000000000000000000

1.5 → 0 01111111 10000000000000000000000

Solution

- Splitting exponents and mantissas, and adding implicit bit

		Sign	Exponent	Mantissa
	7.5 →	0	10000001	1.11100000000000000000000
X	1.5 →	0	01111111	1.10000000000000000000000

The implicit bit is included

Solution

- Multiply: add exponents and multiply mantissas

7.5 →	0	10000001	1.11100000000000000000000
X 1.5 →	0	01111111	1.10000000000000000000000
		+	×
	0	100000000	10.11010000000000000000000

Solution

- Multiply: remove one bias from exponent (there are two)

7.5 →	0	10000001	1.11100000000000000000000
X 1.5 →	0	01111111	1.10000000000000000000000
	0	100000000	10.11010000000000000000000
		- 01111111	
	0	10000001	10.11010000000000000000000

Félix García-Carballeira, Alejandro Calderón Mateos

Solution

- Multiply: normalize result…

	7.5 →	0	10000001	1.11100000000000000000000
X	1.5 →	0	01111111	1.10000000000000000000000
	11.25	0	10000001	10.11010000000000000000000

+1 ← → 1

Solution

- Multiply: normalize result…

7.5 →	0	10000001	1.11100000000000000000000
X 1.5 →	0	01111111	1.10000000000000000000000
11.25	0	10000010	1.01101000000000000000000

Solution

- Eliminate the implicit bit and store the result

11.25 `0 10000010 011010000000000000000000`

IEEE 754 Evolution

- 1985 – IEEE 754
- 2008 – IEEE 754-2008 (754+854)
- 2011 – ISO/IEC/IEEE 60559:2011 (754-2008)

Name	Common name	Base	Digits	E min	E max	Notes	Decimal digits	Decimal E max
binary16	Half precision	2	10+1	−14	+15	storage, not basic	3.31	4.51
binary32	Single precision	2	23+1	−126	+127		7.22	38.23
binary64	Double precision	2	52+1	−1022	+1023		15.95	307.95
binary128	Quadruple precision	2	112+1	−16382	+16383		34.02	4931.77
decimal32		10	7	−95	+96	storage, not basic	7	96
decimal64		10	16	−383	+384		16	384
decimal128		10	34	−6143	+6144		34	6144

http://en.wikipedia.org/wiki/IEEE_floating_point

ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 2 (II)
Floating point

Computer Structure

Bachelor in Computer Science and Engineering