

# CS Basics 2)

## Encoding Numbers

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

# Contents

- ▶ **Binary vs. Text Files**
- ▶ **Endianness**
- ▶ **Negative Numbers**
- ▶ **Introduction to Floating Point**
  - Fixed Point Numbers
  - Floating Point Numbers, IEEE 754
- ▶ **Conclusion**

# Binary vs. Text Files

---

# Binary vs. Text Files

## Different file formats (examples)

- Images: jpeg, gif, png, svg, ...
- Documents: pdf, doc, ppt, xls, ...
- Executables (Programs): exe, dll, so, class, ...
- Plain text: txt, html, tex, ...
- Source code: asm, c, cpp, java, ...

## Two primary types

- *Binary* files (images, documents, executables, ...)
- *Text files* (plain text, source code, ...)

## Distinction not always strict

- `svg` is a format for vector graphics based on XML and can thus be considered a text file
- Office documents are often also a form of compressed XML
- ...

# Text Files

## Different character encodings

- 7 bit / US-ASCII
- ISO-8859-\*
- UTF-8
- ...

## How to work with text files?

- Using any text editor or IDE
- vim, emacs, Notepad (windows), gEdit, kate, sublime, atom, ...
- Eclipse, NetBeans, IntelliJ, kDevelop, ...

# Binary Files I

## A lot of different file formats

- Open and proprietary
- Encoding and structure depend on type!

## Executable files (Programs)

- Windows: `exe`
- GNU/Linux: `elf` (32 bits), `elf64` (64-bit)
- Contain machine instructions and data
- Can be analyzed with a *decompiler* or *disassembler*

## Images

- `jpeg`: format family, lossy compression
- `gif`: 8-bit color, animations possible
- `png`: 32-bit color, no animations, modern lossless compression (good for screenshots)
- `svg`: vector graphics, not a binary file type (see Slide 4)
- Can be manipulated with libraries and image editors

# Binary Files II

## How to analyze any type of binary data?

- Using a *hex editor*
- Editors: Bless, xxd, ...

## Bless functionality

- Display binary and textual representation at the same time
- Various conversions to different representations
- Can read and edit any file – also executable files

# Binary Files III

Hex editor window showing the beginning of an ELF file. The address 00000007 is highlighted, and the bytes 7F 45 4C 46 are selected. The conversion panel below shows the selected bytes in various formats:

|                  |       |                  |                       |              |                      |
|------------------|-------|------------------|-----------------------|--------------|----------------------|
| Signed 8 bit:    | 127   | Signed 32 bit:   | 2135247942            | Hexadecimal: | 7F 45 4C 46          |
| Unsigned 8 bit:  | 127   | Unsigned 32 bit: | 2135247942            | Decimal:     | 127 069 076 070      |
| Signed 16 bit:   | 32581 | Float 32 bit:    | 2.622539E+38          | Octal:       | 177 105 114 106      |
| Unsigned 16 bit: | 32581 | Float 64 bit:    | 1.16843158995565E+305 | Binary:      | 01111111 01000101 01 |

Additional options:  Show little endian decoding,  Show unsigned as hexadecimal, ASCII Text: ELF

Offset: 0x0 / 0x51c Selection: None INS



# Binary Files IV

## Data is ultimately always encoded in binary

### Example

- Capital letter “S” is encoded with 0x53 (ASCII/UTF-8)
- For the computer, this is a set of 8 bits: 01010011
- Can also be part of a different number
  - 0x53 may be interpreted as decimal 83
  - 0x53\_61 may be interpreted as the decimal 21'345
  - 0x53\_61\_6D\_0A may be interpreted as the decimal 1'398'893'834
  - 0x53\_61\_6D\_0A\_77\_61\_73\_0A may be interpreted as the floating point number 4.54365038640977.10<sup>93</sup>
- But: This pattern can also mean anything else
  - Can be part of an instruction
  - Can be any data (object, address, ...)
  - ...

# Endianness

# Endianness

## Having multiple bytes of data...

- Example: 0x53 0x61 0x6D 0x0A 0x77 0x61 0x73 0x0A
- Can be read/interpreted left-to-right (like European languages)

0x53 0x61 0x6D 0x0A 0x77 0x61 0x73 0x0A  
0 1 2 3 4 5 6 7

- ...but can also be read/interpreted right-to-left (like in Arabic)

0x0A 0x73 0x61 0x77 0x0A 0x6D 0x61 0x53  
7 6 5 4 3 2 1 0

- The same data — but different meaning!

## Which number does the above data represent?

- 0x53616D0A7761730A ?
- 0x0A7361770A6D6153 ?

# Big Endian vs. Little Endian

## Big Endian

- Data is stored with the *most significant byte (MSB)* first
- Example: Number 0x20A1 is stored as follows

```
0x20 0xA1  
0    1
```

## Little Endian

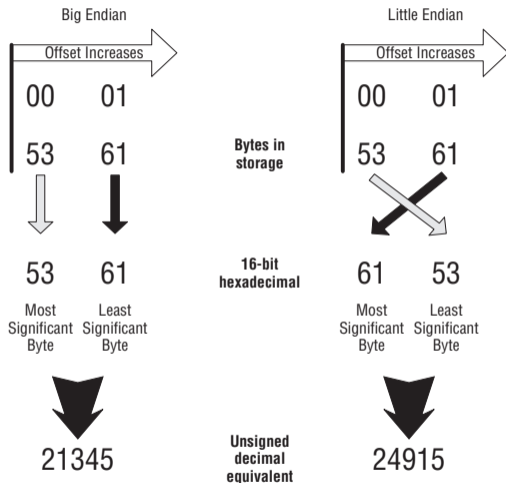
- Data is stored with the *least significant byte (LSB)* first
- Example: Number 0x20A1 is stored as follows

```
0xA1 0x20  
0    1
```

## Endianness is dependent of CPU architecture

- *Intel x86 uses little endian*
- Other architectures may use big endian
- Some CPUs even support both

# Big Endian vs. Little Endian



**Figure 5-5:** Big endian vs. little endian for a 16-bit value

# Negative Numbers

# How to Represent Negative Numbers?

## So far, we have seen positive numbers

- Example: 15
- In memory: 0x0F (8 bit) or 0x0000000F (32 bit)

## What about negative numbers?

- Example: -15
- In memory: ???

# Idea: Sign Bit

## What if we use one bit for the sign?

- Taking the first bit as sign bit...
- 15 is still 0x000F (16 bit)
- -15 would be 0x800F

## Problems

- Two representations of 0 exist: 0x0000 and 0x8000
- Standard (bit-level) addition does not work:

$$15 + (-15) = 0x000F + 0x800F = 0x801E = -30$$



# One's Complement

## One's Complement

- The negative number is obtained by inverting all the bits
- $15 = 0x000F$
- $15 = 0b0000\ 0000\ 0000\ 1111$
- $-15$  is represented by  $0b1111\ 1111\ 1111\ 0000$
- $-15 = 0xFFFF_0$

## Problems

- We *still* have two representations of 0:  $0x0000$  and  $0xFFFF$
- Addition *still* does not work:

$15 = 0x000F$  and  $-5 = 0xFFFFA$ . The sum is  $0x10009 = 9$ , if we ignore the overflow

# Two's Complement

## Negative numbers must respect arithmetic

- $15 + (-5) = 10$
- Idea:  $-X$  is represented by  $2^n - X$ , with  $n$  being the *bit-width*
- $X + (-X) = X + (2^n - X) = 0$  with overflow

## Example 1

- Representation of  $-1$  in 16 bits
- $2^{16} - 1 = 2^{15} + 2^{14} + 2^{13} + 2^{12} + 2^{11} + \dots + 2^1 + 2^0$
- $-1$  is written `0xFFFF`
- $1 + (-1) = 0x10000$  (overflow!)

## Example 2

- Representation of  $-20$  in 16 bits
- $2^{16} - 20 = 65516 = \dots$
- ...hard to compute

# Two's Complement

## Method for easy computation of two's complement

- Take the binary representation of  $X$
- Invert all the bits of  $X$
- Add 1

## Example 2 again

- Want:  $-20$
- $20 = 0b0000\ 0000\ 0001\ 0100$
- ...inverting all the bits
- $0b1111\ 1111\ 1110\ 1011$
- ...adding 1
- $0b1111\ 1111\ 1110\ 1100 = 0xFFEC$

## Arithmetic works!

- $15 + (-5) = 10$
- $0b0000\ 1111 + 0b1111\ 1011 = 0b1\ 0000\ 1010$
- Ignoring overflow:  $0b0000\ 1010 = 10$

# Introduction to Floating Point

# Floating Point Arithmetic

In the early days, CPUs were only able to handle integer arithmetic.

- However, for scientific and also everyday purposes, *real numbers* ( $\mathbb{R}$ ) are required.
- The data types for working with real numbers have special properties that every programmer must know.

Representation of real numbers and *floating point arithmetic* is standardized in the *IEEE 754 standard*, which is supported by most programming languages and CPUs nowadays.

# Fixed Point Numbers

How to represent the decimal number 543.125 in *fixed point* arithmetic? Recall: Every number in base  $b$  can be written as

$$x = \sum_{i=-\infty}^{\infty} d_i b^i$$

where  $d_i$  are the respective digits of the number and  $b$  is the base.

This way, the number 543.125 may be written as the sum:

$$543.125 = 5 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

# Binary Fixed Point Numbers

As the number 543.125 exists *independently* of any base, it must be possible to obtain a binary representation. We proceed as follows:

1. First, the integral part of the number is converted to binary

$$543 = 0b10\ 0001\ 1111$$

2. Then, the fractional part of the number is converted as well

$$0.125 = 0b0.001$$

The whole, fixed point binary number is then

$$543.125 = 0b10\ 0001\ 1111.001$$

# Converting to Fixed Point Binary I

First, we convert the *integer* part as usual, using division by 2 with remainder:

**Example: 543**

|     |  |   |
|-----|--|---|
| 543 |  | 1 |
| 271 |  | 1 |
| 135 |  | 1 |
| 67  |  | 1 |
| 33  |  | 1 |
| 16  |  | 0 |
| 8   |  | 0 |
| 4   |  | 0 |
| 2   |  | 0 |
| 1   |  | 1 |

543 = 0b10 0001 1111



## Converting to Fixed Point Binary II

To convert the *fractional* (or *decimal*) part, we do the opposite and multiply by two until we have no remainder:

**Example: 0.125**

0.125 | 0.

0.25 | 0

0.5 | 0

1 | 1

0.125 = 0b0.001

# Some Mathematical Considerations

A rational number ( $\mathbb{Q}$ ) is either finite or infinite periodic.

- The property of “rationality” is independent of the base used to represent a number.

A rational number may have a finite representation in one base and a periodic infinite representation in another base.

- Example: the number  $\frac{1}{3}$  has an infinite periodic representation in base 10...
- ...but is written  $0.1_3$  in base 3!

Irrational numbers ( $\mathbb{R} \setminus \mathbb{Q}$ ) are infinite and non-periodic.

- They do not have a finite representation *in any* base.

# Infinite Periodic Number Example

0.1 is finite in decimal — but not in binary:

0.1 | 0.  
0.2 | 0  
0.4 | 0  
0.8 | 0  
1.6 | 1 -> 0.6  
1.2 | 1 -> 0.2  
0.4 | 0  
0.8 | 0  
1.6 | 1 -> 0.6  
1.2 | 1 -> 0.2  
...

$$0.1 = 0b0.000110011\overline{0011}$$

# Scientific Notation

- The main drawback of fixed point numbers is that their size depends on their magnitude and their precision.
- In engineering, most of the time, the precision can be reduced as the magnitude increases. This is the concept of *significant digits*.
- On pocket calculators, this is usually known as “scientific notation”. Examples:
  - $1.344 \cdot 10^4$  (= 13 440)
  - $2.342 \cdot 10^{-5}$  (= 0.000 023 42)
  - $4.430 \cdot 10^0$  (= 4.430)
- All these numbers have 4 significant digits (called the *mantissa*). The *exponent* ( $10^x$ ) “scales” the numbers.
- With 4 significant digits, the mantissa is always between 1.000 and 9.999.

# The IEEE 754 Standard

The IEEE 754 standard used for binary representation of floating point numbers is based on the same idea:

- The mantissa is a number  $\geq 1.0$  and  $< 2.0$ .
- The exponent is now an exponent of 2. It may take positive or negative values.
- Maximum exponent and *precision* (number of significant digits in the mantissa) depends on the encoding format, see Slide 36.
- The standard also introduces special values (negative infinity, positive infinity, not a number, zero).
- It also contains rules for rounding.

# IEEE 754, An Example

Example: Encoding the decimal number  $0.5 (= 1.0 \cdot 2^{-1})$  in IEEE 754:<sup>1</sup>

$0.5 = 0 \ 0111 \ 1110 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 000$

- The first bit is the **sign bit**. Its value is **0** for positive numbers and 1 for negative numbers.
- The **exponent**, in this case  $2^{-1}$ , is encoded with a so-called **bias**.
- The **mantissa** is **1** (not encoded due to the **hidden bit**).

---

<sup>1</sup>The number is encoded in single precision format (e.g. `float` in C or Java).

# IEEE 754, The Exponent

- The exponent can be positive or negative.
- It is encoded as a signed integer with **bias**, i.e. without using two's complement!
- The bias is added to the original exponent and corresponds to half the magnitude of the exponent *minus one*.  
For example, if the exponent is 8 bits, its magnitude is  $2^8 = 256$  and the bias is  $2^{(8-1)} - 1 = 127$ .
- Two exponents,  $00_{16}$  and  $FF_{16}$ , are reserved to increase precision for very small numbers and for representing special values.

# IEEE 754, The Mantissa

- Contains a value in the interval  $1 \leq \text{mantissa} < 2$ .
- As the mantissa is always  $\geq 1$ , the bit for the 0th power (i.e.  $2^0$ ) may be omitted, increasing precision. This is called the **hidden bit**. Due to this, the mantissa is 0 in the given example.
- Exception: For **denormalized numbers** (exponent 0), the mantissa  $m$  is  $0 < \text{mantissa} < 1$ ! This is used for very small numbers (below  $2^{-126}$ ).
- The bits of the mantissa represent the sum of negative powers of 2.



## IEEE 754, Values

The following table lists possible values for IEEE 754 encoded numbers:

| Exponent               | Mantissa | Value                     | Description        |
|------------------------|----------|---------------------------|--------------------|
| $00_{16}$              | $= 0$    | $0$                       | Zero               |
| $00_{16}$              | $> 0$    | $\pm 0.m \cdot 2^{(1-b)}$ | Denormalized       |
| $01_{16}$ to $FE_{16}$ | any      | $\pm 1.m \cdot 2^{(e-b)}$ | Normalized         |
| $FF_{16}$              | $= 0$    | $\pm \infty$              | $\pm$ Infinity     |
| $FF_{16}$              | $> 0$    | NaN                       | Not a Number (NaN) |

Remarks:

- $e$  is the exponent,  $b$  the bias value.
- There are two possible representations for the value 0 (+0 and -0).
- Exponents  $00_{16}$  and  $FF_{16}$  represent special values!
- All other exponents ( $01_{16}$  to  $FE_{16}$ ) are *normalized* values.

# Comments on Special Values

- The value zero needs a special representation as it is not possible to represent it using the standard encoding (see example above).
- Infinity represents numbers whose magnitude cannot be encoded. However, arithmetical operations where one operand is infinity are well defined by IEEE 754 (see next slide).
- Operations where one operand has the value NaN cause an error.
- Warning: Handling of denormalized values is hardware and implementation dependent and may lead to performance issues. This must be considered when using them.

# IEEE 754

## Operations with Special Values

| Operation                     | Result       |
|-------------------------------|--------------|
| $x / \pm \infty$              | 0            |
| $\pm \infty \cdot \pm \infty$ | $\pm \infty$ |
| $\pm \text{non zero} / 0$     | $\pm \infty$ |
| $\pm 0 / \pm 0$               | NaN          |
| $\infty + \infty$             | $\infty$     |
| $\infty - \infty$             | NaN          |
| $\pm \infty / \pm \infty$     | NaN          |
| $\pm \infty \cdot 0$          | NaN          |

# IEEE 754 Formats

IEEE 754 (2019) defines three *basic formats* for binary floating point numbers:<sup>2</sup> *Single precision*, *double precision* and *quadruple precision*. They differ only by the number of bits required to store them:

|                               | Sign | Exponent | Mantissa | Bias  |
|-------------------------------|------|----------|----------|-------|
| Single precision (32 bit)     | 1    | 8        | 23       | 127   |
| Double precision (64 bit)     | 1    | 11       | 52       | 1023  |
| Quadruple precision (128 bit) | 1    | 15       | 112      | 16383 |

Note: Due to the hidden bit, precision equals to mantissa +1.

---

<sup>2</sup>Formats with different bit widths, as well as decimal formats (using radix 10) are also specified as non-basic formats and are not treated here.

# IEEE 754 Rounding

IEEE 754 has five different rounding modes. The first two round to nearest, the others are *directed roundings*.

- **Round to nearest, ties to even**  
Round to nearest encodable value. If exactly midway, round to the nearest value where the least significant bit of the mantissa is 0.
- **Round to nearest, ties away from zero**  
Round to nearest also, but round up (positive numbers) or down (negative numbers)
- **Round toward 0**  
Rounds to 0 (*truncation*)
- **Round toward  $+\infty$**   
Always round *up*.
- **Round toward  $-\infty$**   
Always round *down*.

## Examples for Rounding

| Rounding Mode                   | +11.5 | +12.5 | -11.5 | -12.5 |
|---------------------------------|-------|-------|-------|-------|
| to nearest, ties to even        | +12.0 | +12.0 | -12.0 | -12.0 |
| to nearest, ties away from zero | +12.0 | +13.0 | -12.0 | -13.0 |
| toward 0                        | +11.0 | +12.0 | -11.0 | -12.0 |
| toward $+\infty$                | +12.0 | +13.0 | -11.0 | -12.0 |
| toward $-\infty$                | +11.0 | +12.0 | -12.0 | -13.0 |

# Comparing Floating Points Values

Many floating point operations are not associative due to rounding! That is, the same expression, but *computed in a different order*, may create different results!

This makes the comparison of floating point values particularly tricky and complicated.

Moreover, different compilers, even though implementing the same standard, may yield different results!

## Problem: Absorbtion (C)

Absorption occurs when working with numbers with large differences in magnitude:

```
#include <stdio.h>

int main(void) {
    float val1 = 100.0, val2 = 0.05, val3 = 0.05;

    printf("sum 1: %f\n", (val1 + val2) + val3);
    printf("sum 2: %f\n", val1 + (val2 + val3));
}
```

```
> ./absorbtion
sum 1 : 100.100006
sum 2 : 100.099998
```



# Problem: Absorbtion (Java)

```
public class Absorbtion {
    public static void main (String args[]) {
        float val1 = 100.0f;
        float val2 = 0.05f;
        float val3 = 0.05f;

        System.out.println("sum 1 : " + ((val1 + val2) + val3));
        System.out.println("sum 2 : " + (val1 + (val2 + val3)));
        System.out.println("sum 3 : " + (val1 + val2 + val3));
    }
}
```

```
java Absorbtion
sum 1 : 100.100006
sum 2 : 100.1
sum 3 : 100.100006
```

## Comparing Floats: Idea

⚠ Due to such problems, it is totally “forbidden” to compare floating point values using the == operator.

This solution — seen in many books — fails most of the time:

```
final float EPSILON = 10e-6;
...
if Math.abs(value1 - value2) < EPSILON {
    ...
}
```

Problem: The value of the constant EPSILON depends on the magnitude of the tested values.

# Comparing Floats: Bitwise I

One can try to compare both values bit-by-bit. For example:

1. Transform the float values **⚠** bitwise into integers (e.g. in Java using the method `floatToIntBits()`)
2. Mask the 2 LSB of each value
3. Compare the result.

Question: **⚠** Why does this comparison fail?

# Comparing Floats: Bitwise II

```
#include <stdio.h>
typedef union {
    int intVal;
    float floatVal;
} intFloat;

int main(void) {
    intFloat testVal;

    testVal.intVal = 0x3fffffff;
    printf("v1 as int : %x\n", testVal.intVal);
    printf("v1 as float : %f\n", testVal.floatVal);

    testVal.intVal = 0x40000001;
    printf("v2 as int : %x\n", testVal.intVal);
    printf("v2 as float : %f\n", testVal.floatVal);
    return 0;
}
```

## Comparing Floats: Bitwise III

```
./a.out  
v1 as int : 3fffffff  
v1 as float : 2.000000  
v2 as int : 40000001  
v2 as float : 2.000000
```

# Comparing Floats: Absolute Error I

- The main idea is to have an error margin which depends on the magnitude of the two values being compared.
- The simplest formula for that is:

$$\frac{|a - b|}{b} < \varepsilon$$

for a given  $\varepsilon$  value independent of the magnitude of  $a$  and  $b$ .

- **⚠** If  $b$  is zero (or very near to zero) this direct computation fails!

## Comparing Floats: Absolute Error II

```
diff ←  $| |a| - |b| |$   
if a is strictly equal to b then  
    return true      {for  $\infty$  and NaN}  
else if  $a = 0$  or  $b = 0$  or  $(|a| + |b| < \text{min normal value})$  then  
    {for very small value of a and b}  
    return  $\text{diff} < (\epsilon * \text{min normal value})$   
else  
    return  $(\text{diff} / \min(|a| + |b|, \text{max float value})) < \epsilon;$   
end if
```

Note: *min normal value* is the smallest value which can be stored *without* using a denormalized value.

## Total ordering predicate

The standard provides a predicate `totalOrder` which defines a total ordering for all floating point numbers in each format.

The predicate agrees with the normal comparison operations when they say one floating point number is less than another.

The normal predicate compares `-0` and `+0` as equal.

The normal comparison operations however treat NaNs as unordered.



# Conclusion

# Conclusion

## Encoding Numbers

- Little and big endian,
- Least and most significant bytes (MSB/LSB),
- Signed numbers are encoded using 2's complement.

## Real numbers

- Are composed of sign, mantissa and exponent.
- Equality (“==” does not work) and comparison is tricky.