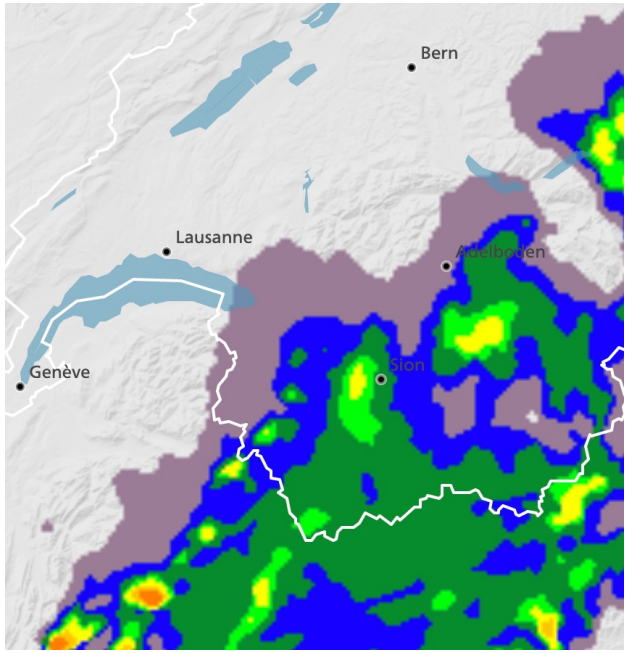
Information, Computation, and Communication

Representation of Information

Computation Works with Information



Scientific computation
→*numbers*

Control process
→*signals (measurements, control…)*

Data Centers Information management
→*text, photos, movies…*

Objectives

- In which ways can we represent numbers and letters?

- Is it possible to build an exact representation of the real world?

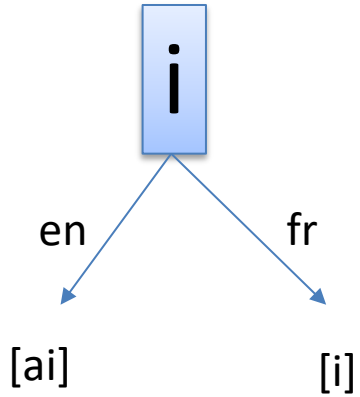
Agenda

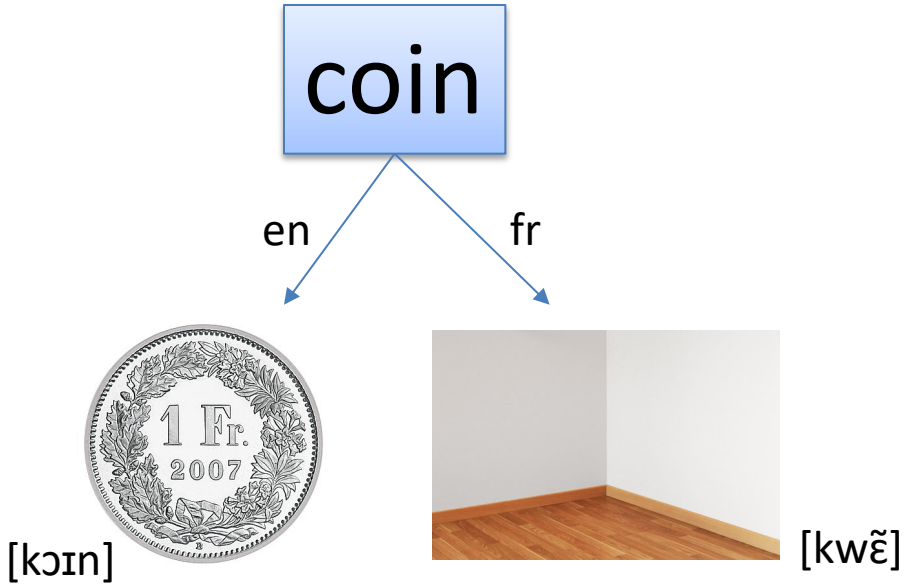
- Representation of the information

- Representation of
  - Natural Numbers (e.g., 2 4 5 6): operations/domain
  - Integers (e.g., -1 -5 4 45698)
  - Reals (e.g., 3.4 4.756): fix and floating point
  - Alphabets and Images

Convention

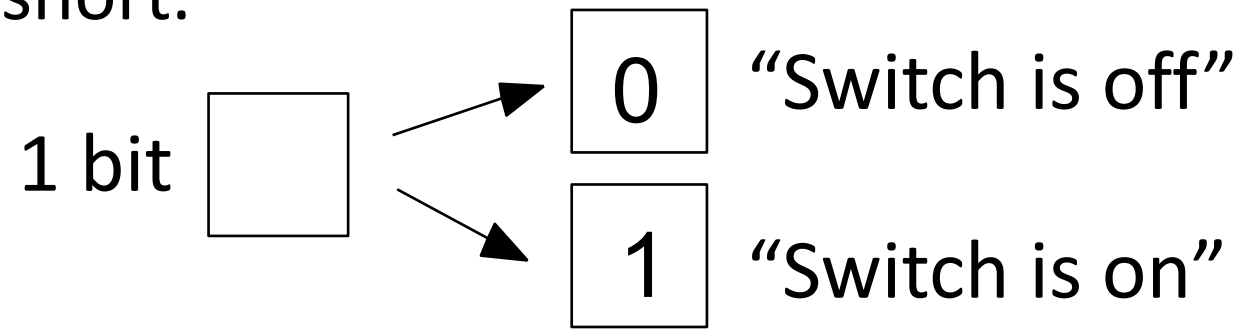
- A **representation** is a convention (an agreement between people about the meaning of symbols)
- Sets of symbols in use: ten digits (0-9), 26 letters (a-z), seven Roman numbers, 4000+ Chinese keys,..
- **Representation = Symbols + Interpretation**





Binary System and Bit

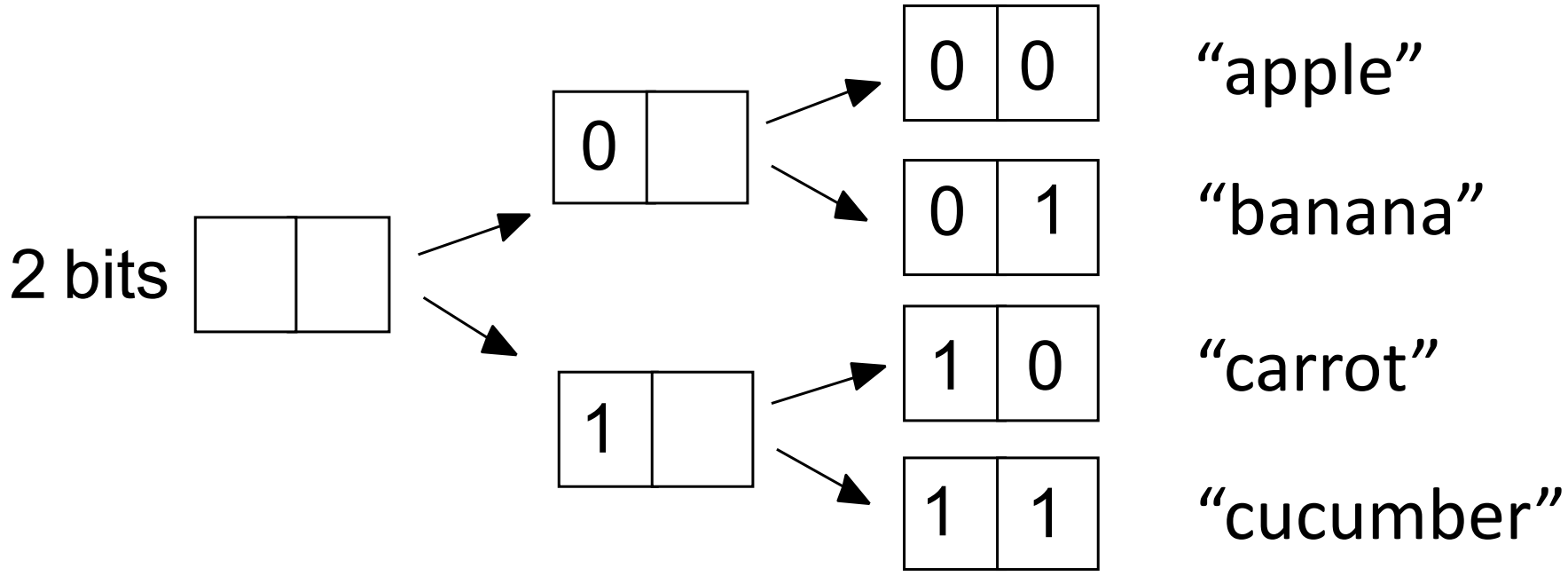
- **Minimal number** of symbols needed is **two.**
- All information can be represented with the help of a set of **binary elements.**
- By convention, we use the **symbols 0 and 1** to represent the value of a binary element.
- Such a binary element is called **binary digit** or **bit (b)** in short.



1 bit can distinguish 2^1 distinct pieces of information

More than Two Distinct Pieces of Information

- We want to distinguish several objects. How can we represent the four objects below?
- We use a sequence of two bits:



2 bits can distinguish 2^2 distinct pieces of information

Information Stored in Sequence of Bits

n bits can distinguish 2^n distinct pieces of information

- To represent **k distinct pieces** of information we need at **$\lceil \log_2(k) \rceil$ bits**, where $\lceil x \rceil$ arounds x to the closest integer that is higher or equal to x. If $k=2^n$, then we need precisely n bits because $\log_2(2^n) = n \log_2(2) = n$.
- How many bits do we need to represent
  - the 7 days of a week? $7 < 8=2^3 \Rightarrow$ 3 bits
  - the 10 digits? $10 < 16=2^4 \Rightarrow$ 4 bits
  - the 26 letters? $26 < 32=2^5 \Rightarrow$ 5 bits

$k \leq 2^n$
$\log_2(k) \leq \log_2(2^n)$
$\log_2(k) \leq n$
Pick n as the first integer larger than $\log_2(k)$

n bits can distinguish 2^n distinct pieces of information

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
10	1'024
20	1'048'576
30	1'073'741'824
32	4'294'967'296

Good practice for fast estimation:

2^{10} = Kb (Ki) $\approx 10^3$ = kilo (k)

2^{20} = Mb (Mi) $\approx 10^6$ = mega (M)

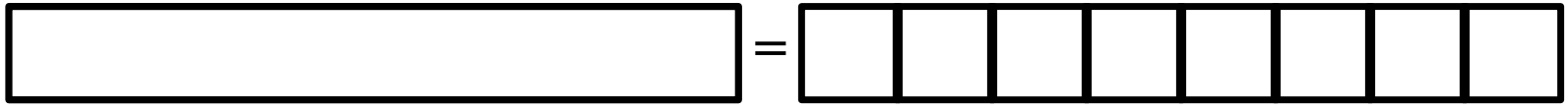
2^{30} = Gb (Gi) $\approx 10^9$ = giga (G)

How many elements can be approx. distinguished with 32 bits?

$2^{32} = 2^{30+2} = 2^{30} \cdot 2^2 \approx 4\ G = 4 \cdot 10^9$ elem.

8 Bits = Byte

- **By convention**,
  a **sequence of 8 bits** is called a **byte (octet in French).**
- The shortcut for byte is B.
- Recall the short of bit is b.



- How many distinct pieces of information can be stored in a byte?

$$2^8 = 256$$

Representation of Numbers

- All numbers (and, therefore, letters, sentences, etc.) can be represented with a **sequence of binary digits**.
- **Definition**: a sequence of 0's and 1's is called **binary pattern**, e.g., 0100010111
- We give **meaning** to a binary pattern by providing an **interpretation method**.
- Next, we will see different interpretation methods: one for **natural numbers**, one for **integers**, one for **reals**, etc.

Example of Different Interpretations

- The binary pattern 100101 can have different meaning:
  - 37 (interpreted as natural number without sign)
  - -5 (interpreted as number with sign)
  - 4.635 (interpreted as a fixed-point number)
  - 26 (interpreted as a floating-point number)
  - …

Agenda

- Representation of the information

- Representation of
  - **Natural Numbers (e.g., 2 4 5 6): operations/domain**
  - Integers (e.g., -1 -5 4 45698)
  - Reals (e.g., 3.4 4.756): fix and floating point
  - Alphabets and Images

How to Represent Natural Numbers?

- Recall, natural number are non-negative integers (i.e., 0,1,2,3….)
- Recall, we need an **interpretation method** to assign **meaning** (a number) to a binary pattern.
- Which natural number should 0100110 represent?

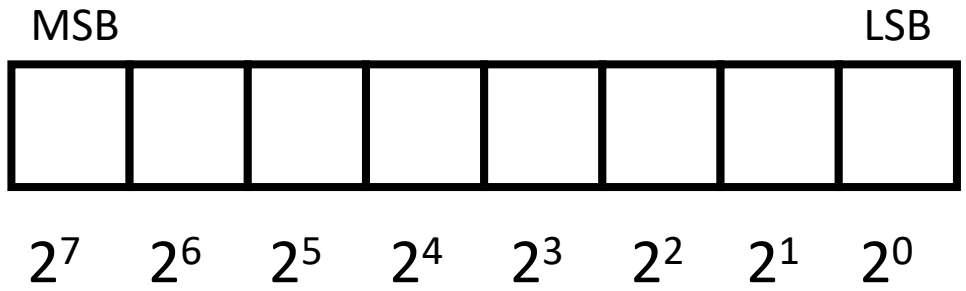
- **One solution**: use the **positional notation** (aka **place-value notation**)

Positional Notation of Numbers

- Example of an integer in base 10: 703
  - The number 703 is the **abbreviated** notation of the expression: $7 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$
- The digit on the **right** is always multiplied by the base (10) raised to the **power 0**
- The power of the base **increases by one** from digit to digit, going from right to left
- This convention of positional notation can be used with **any base**.

Positional Representation in Base 2

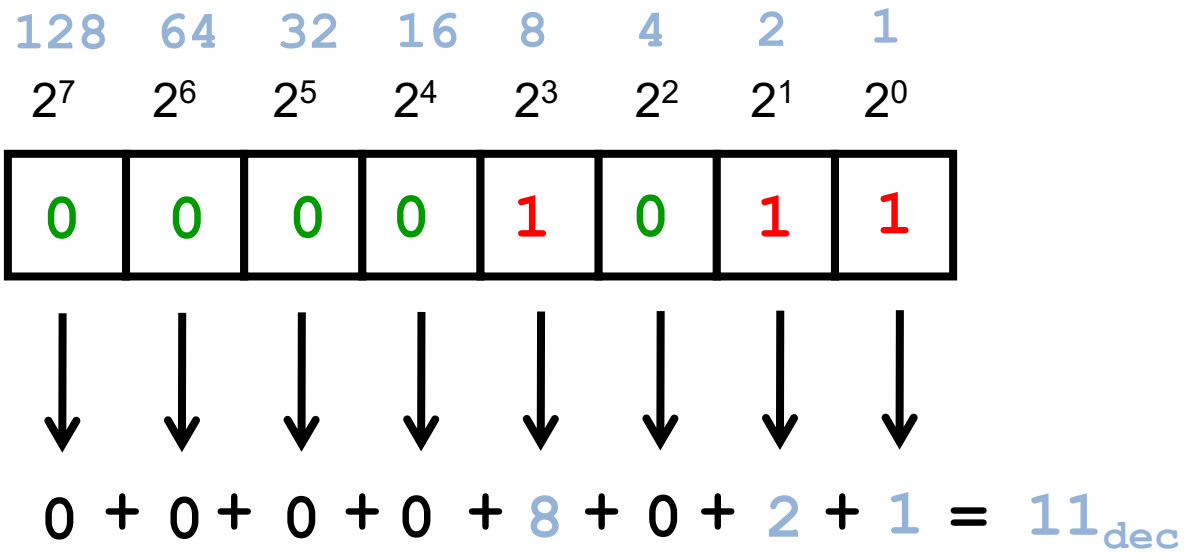
- Uses the same conventions as in base 10 (decimal)



- **Convention**:
  - **M**ost **s**ignificant **b**it (MSB) on the left (multiplied by 10^{n-1})
  - **L**east **s**ignificant **b**it (LSB) on the right (multiplied by $10^0=1$)

Conversions: Binary to Decimal

- Sum up the powers of two that are present in the binary pattern



Conversions: Decimal to Binary

- **Idea:** decompose the decimal number into a sum of powers of two, e.g., $11_{10} = 2^3 + 2^1 + 2^0 = 1011_2$
- **Algorithm:** repetitive division by 2

Base

			b_i		
11	div 2 =	5	(+ 1	rest)	$\Rightarrow 2^0$
5	div 2 =	2	(+ 1	rest)	$\Rightarrow 2^1$
2	div 2 =	1	(+ 0	rest)	$\Rightarrow 2^2$
1	div 2 =	0	(+ 1	rest)	$\Rightarrow 2^3$

$11_{10} = 1011_2$

$$
\begin{aligned}
11 &= 2 \cdot 5 + 1 \\
&= 2 \cdot (2 \cdot 2 + 1) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot 1 + 0) + 1) + 1 \\
&= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1011_2
\end{aligned}
$$

Computation with Binary Numbers: Addition

- Decimal addition

```
    1   5   7
+   2   6   5
-------------
```

Carry

- Binary addition

```
        1   1
+       1   1
-------------
```

Carry

- Addition tables in decimal

1+1= 2	2+1= 3	3+1= 4	4+1= 5
1+2= 3	2+2= 4	3+2= 5	4+2= 6
1+3= 4	2+3= 5	3+3= 6	4+3= 7
1+4= 5	2+4= 6	3+4= 7	4+4= 8
1+5= 6	2+5= 7	3+5= 8	4+5= 9
1+6= 7	2+6= 8	3+6= 9	4+6=10
1+7= 8	2+7= 9	3+7=10	4+7=11
1+8= 9	2+8=10	3+8=11	4+8=12
1+9=10	2+9=11	3+9=12	4+9=13

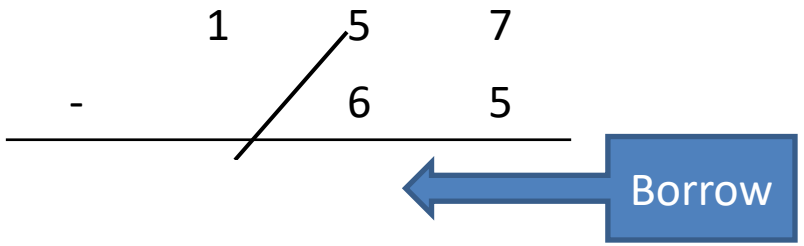
...

- Addition table in binary

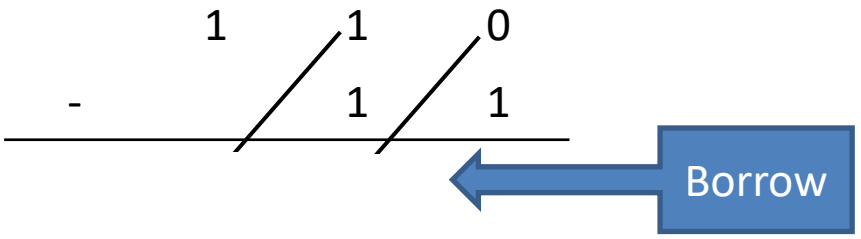
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 10
1+1+1 = 11

Computation: Subtraction

- Decimal subtraction



- Binary substraction



- Substraction tables in decimal

1-1=0	2-2=0	3-3=0	4-4=0
2-1=1	3-2=1	4-3=1	5-4=1
3-1=2	4-2=2	5-3=2	6-4=2
4-1=3	5-2=3	6-3=3	7-4=3
5-1=4	6-2=4	7-3=4	8-4=4
6-1=5	7-2=5	8-3=5	9-4=5
7-1=6	8-2=6	9-3=6	10-4=6
8-1=7	9-2=7	10-3=7	11-4=7
9-1=8	10-2=8	11-3=8	12-4=8
10-1=9	11-2=9	12-3=9	13-4=9

- Substraction table in binary

  0 – 0 = 0

  1 – 0 = 1

  1 – 1 = 0

  10 – 1 = 1 (in decimal 2 – 1 = 1)

  11 – 1 = 10 (in decimal 3 – 1 = 2)

Computation: Multiplication

- Decimal multiplication

1	2	3	2	·	3	2
3	6	9	6			
	2	4	6	4		
3	9	4	2	4		

- Addition tables in decimal

1 x 1 = 1	2 x 1 = 2	3 x 1 = 3
1 x 2 = 2	2 x 2 = 4	3 x 2 = 6
1 x 3 = 3	2 x 3 = 6	3 x 3 = 9
1 x 4 = 4	2 x 4 = 8	3 x 4 = 12
1 x 5 = 5	2 x 5 = 10	3 x 5 = 15
1 x 6 = 6	2 x 6 = 12	3 x 6 = 18
1 x 7 = 7	2 x 7 = 14	3 x 7 = 21
1 x 8 = 8	2 x 8 = 16	3 x 8 = 24
1 x 9 = 9	2 x 9 = 18	3 x 9 = 27
1 x 10 = 10	2 x 10 = 20	3 x 10 = 30

...

- Binary multiplication

	1	0	1	1	·	1	1	0
	1	0	1	1				
		1	0	1	1			
			0	0	0	0		
1	0	0	0	0	1	0		

- Multiplication table in binary

  0 · 0 = 0

  0 · 1 = 0

  1 · 0 = 0

  1 · 1 = 1

Computation: Multi./Division by Multiple of Base

- Multiplication by 10^x in base 10

  1024453 · 100 =
  102445300

- Division by 10^x in base 10

  1024453 : 100 =
  10244 (53 rest)

- Multiplication by 2^x in base 2

  10111011101 · 100 =
  1011101110100
  (Multiplication by 4)

- Division by 2^x in base 2

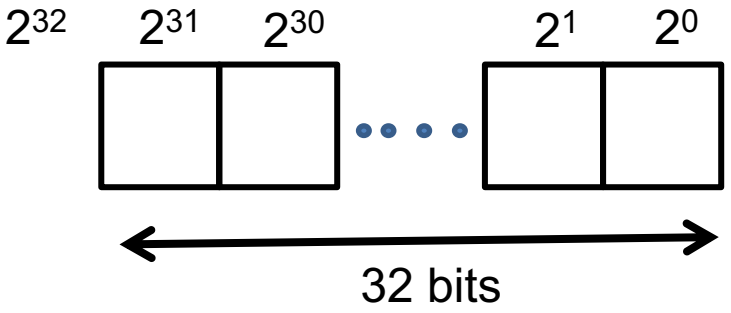
  10111011 : 100 =
  101110 (+ 11 rest)
  (Division by 4)

Capacity

- The **capacity** determines how many and which items we can represent.
- All computing devices work with a fixed capacity, e.g., a **32-bit computer** has instructions that implement basic operations (like addition, multiplication, etc.) rapidly for numbers represented with 32 bits.
- How **many items** can we represent with 32 bits?
- **Covered Domain**: which natural numbers can we represent with a given number of bits?

Natural Numbers: Covered Domain (1)

- If we represent a binary pattern as natural number using the **positional representation in base 2**, the **covered domain for 32 bits is 0 to $2^{32} - 1$**
- Smallest number: 0 (binary pattern with all 0s)
- Largest number: $2^{32} - 1$ (binary pattern with 1s)



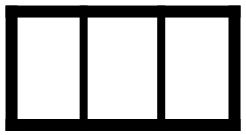
Note: $2^{32} - 1 + 1 = 2^{32}$

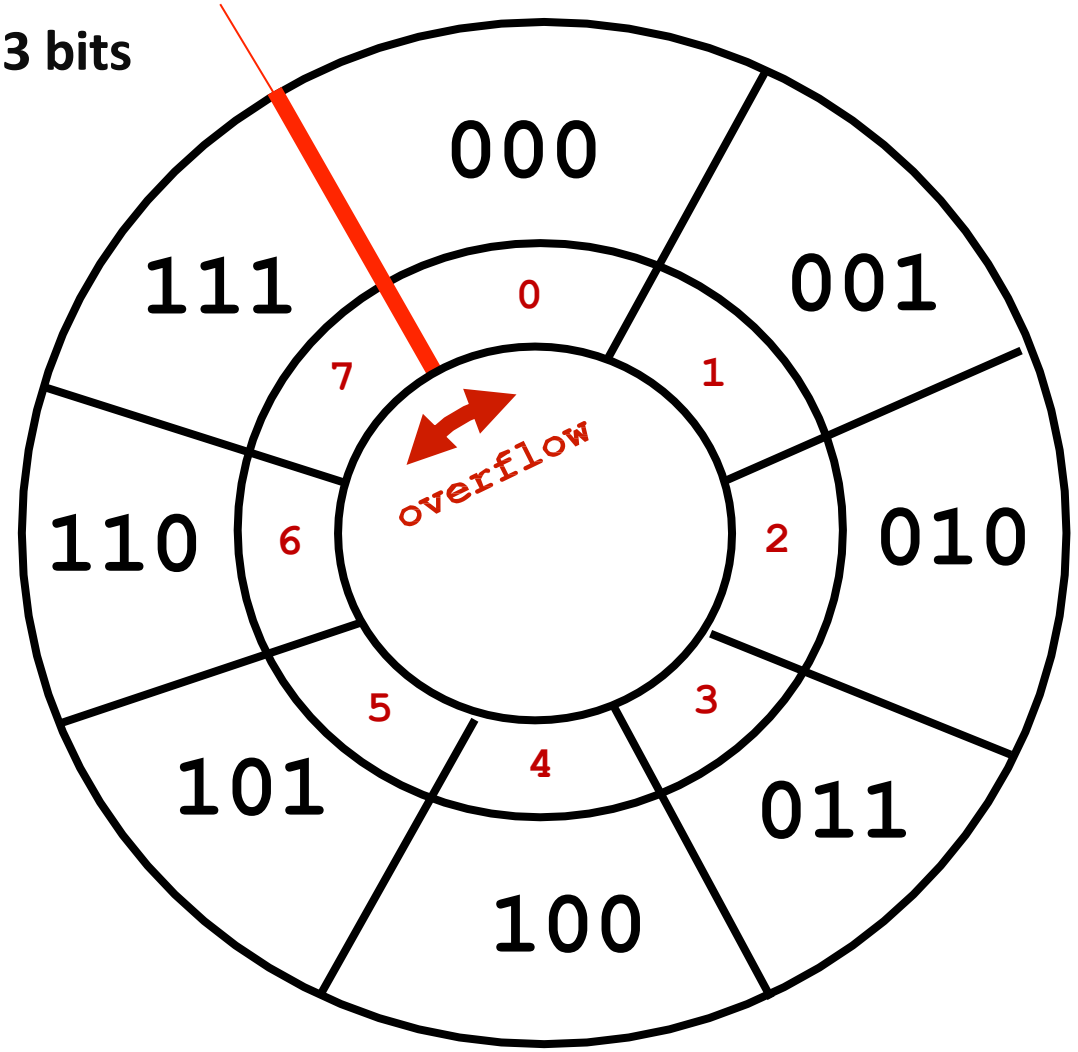
Natural Numbers: Covered Domain (2)

- Computations using this representation are **correct** if the desired result is a **natural number** and belongs to the **covered domain**.
- **Reasons** for results **outside of covered domain**:
  - Integer division: loss of fractional part
  - Multiplication, addition, subtraction: propagation of the carry beyond 2^{31}

Unsigned Integers: Covered Domain and Overflow

Example with 3 bits





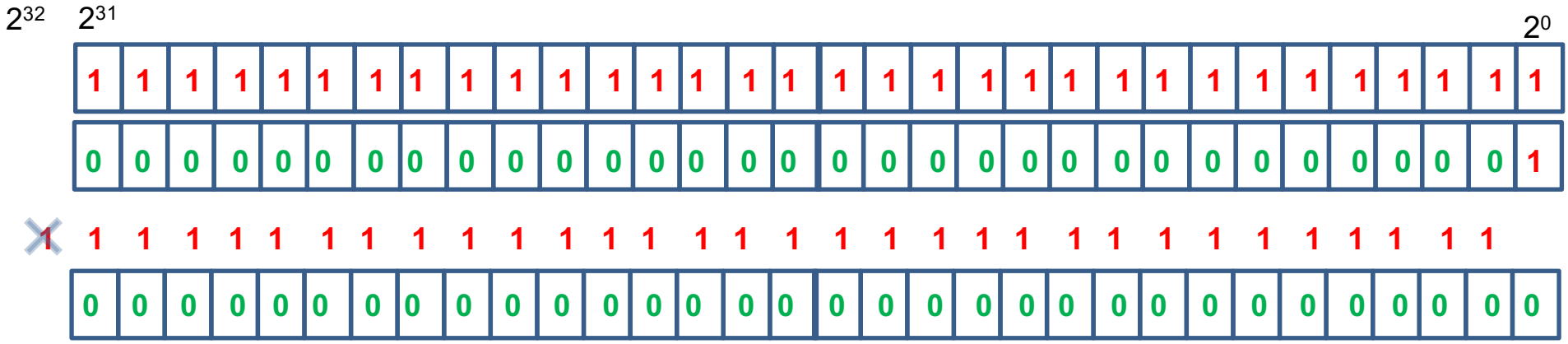
7 + 1 = 0

	1	1	1
+	0	0	1
1	1	1	
	0	0	0

Example of Capacity Overflow

Addition with 32 bits

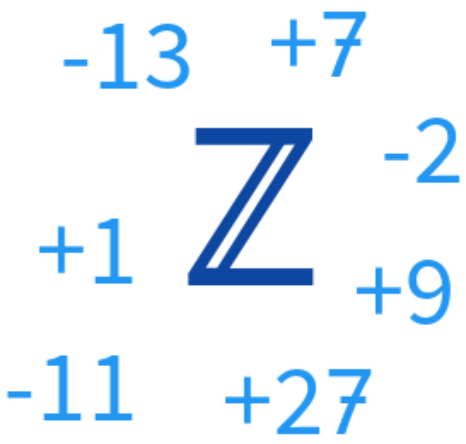
$$(2^{32} - 1) + 1 = 0$$



Agenda

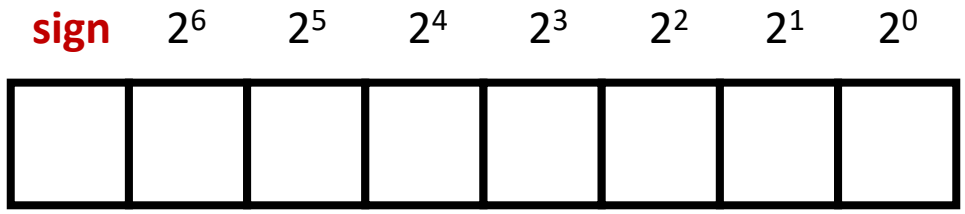
- Representation of the information

- Representation of
  - Natural Numbers (e.g., 2 4 5 6): operations/domain
  - **Integers (e.g., -1 -5 4 45698)**
  - Reals (e.g., 3.4 4.756): fix and floating point
  - Alphabets and Images



Version 1: Representation with Sign and Absolute Value

- The **sign** of a number has two states (**+** or **–).**
  - **One bit** is enough to represent it. **Conventions**: **0 for + , 1 for –**
- What are the consequences of representing a signed number with a **sign** and **absolute value** (using 8 bits)?



  - **Pros**: perfect symmetry of covered domain, two representations for 0 (+/- 0) useful in numerical analysis
  - **Cons**: two representations for 0 and subtraction **cannot** happen by adding an opposite sign number

E.g., representation(1) + representation(-1) ≠ representation(0)

00000001 + 10000001 = 10000010 = representation(-2)

Version 2: Representation using Capacity Overflow

- We aim for a representation of signed integers that allows **subtracting by adding the opposite-sign number**?
- **Reminder**: n bits allow the representation of 2^n numbers (using positional representation in base 2).
  These numbers range from 0 to $2^n - 1$.
- The value 2^n itself cannot be represented with n bits, i.e.,

  $(2^n - 1) + 1 = 2^n$ (in theory)

  But $(2^n - 1) + 1 = 0$ (with n bits)
- **Consequence**: the binary pattern of $(2^n - 1)$ is a good representation of -1 because we obtain 0 when we add 1!

Representation of Signed Integers

- **Properties to verify**: if a is the opposite of b, then

  (1) a + b = 0

  (2) – (–a) = a

- With a capacity of n bits,

  the opposite of a number x is **2^n – x**,

  which is called the **Two's complement of x**.

- **Verification:**

  (1) a + b = (2^n – b) + b = 2^n = 0 (with n bits)

  (2) – (–a) = 2^n – (2^n – a) = a

Using the Two's Complement

- Assume we represent a number with 3 bits
  1 leading sign bit + 2 bits for the value
- Assume we want to represent the number -3:
  - Two's complement of 3 (in decimal) is $2^3 - 3 = 5$.
    - 5 in binary is 101 ($=2^2+2^0$)
  - So, with the Two's Complement interpretation, the binary pattern 101 represents -3.
  - Recall, with the unsigned interpretation 101 represents 5.

Fast Computation of Two's Complement

- In order to represent –x with n bits, we need to compute $2^n - x$
- Note that $2^n - x = 2^n (-1 + 1) - x = ((2^n - 1) - x) + 1$
- $((2^n - 1) - x)$ is called the **One's Complement** and is **very easy to compute**!

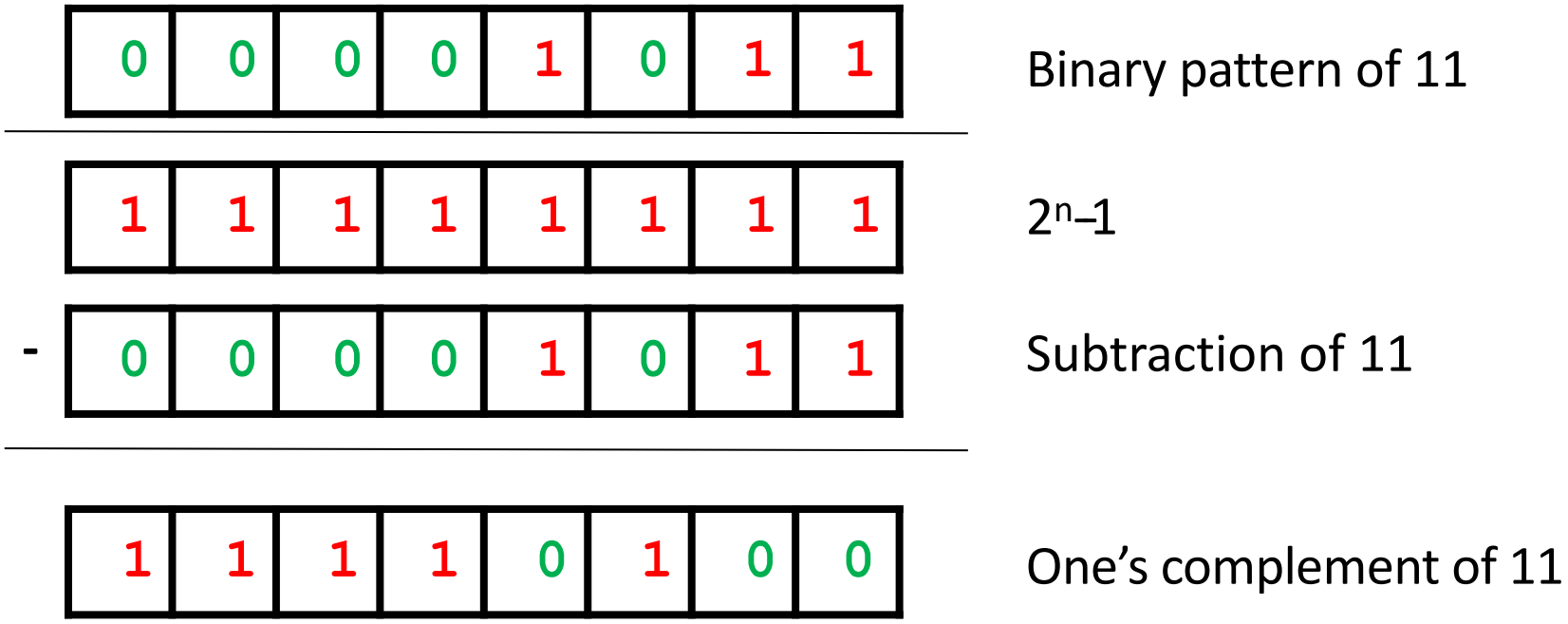
	1	1	1	1	1	1	1	1	$2^n - 1$
-	0	0	1	0	1	0	0	1	x
	1	1	0	1	0	1	1	0	$(2^n - 1) - x$

One just needs to **invert every bit** of x.

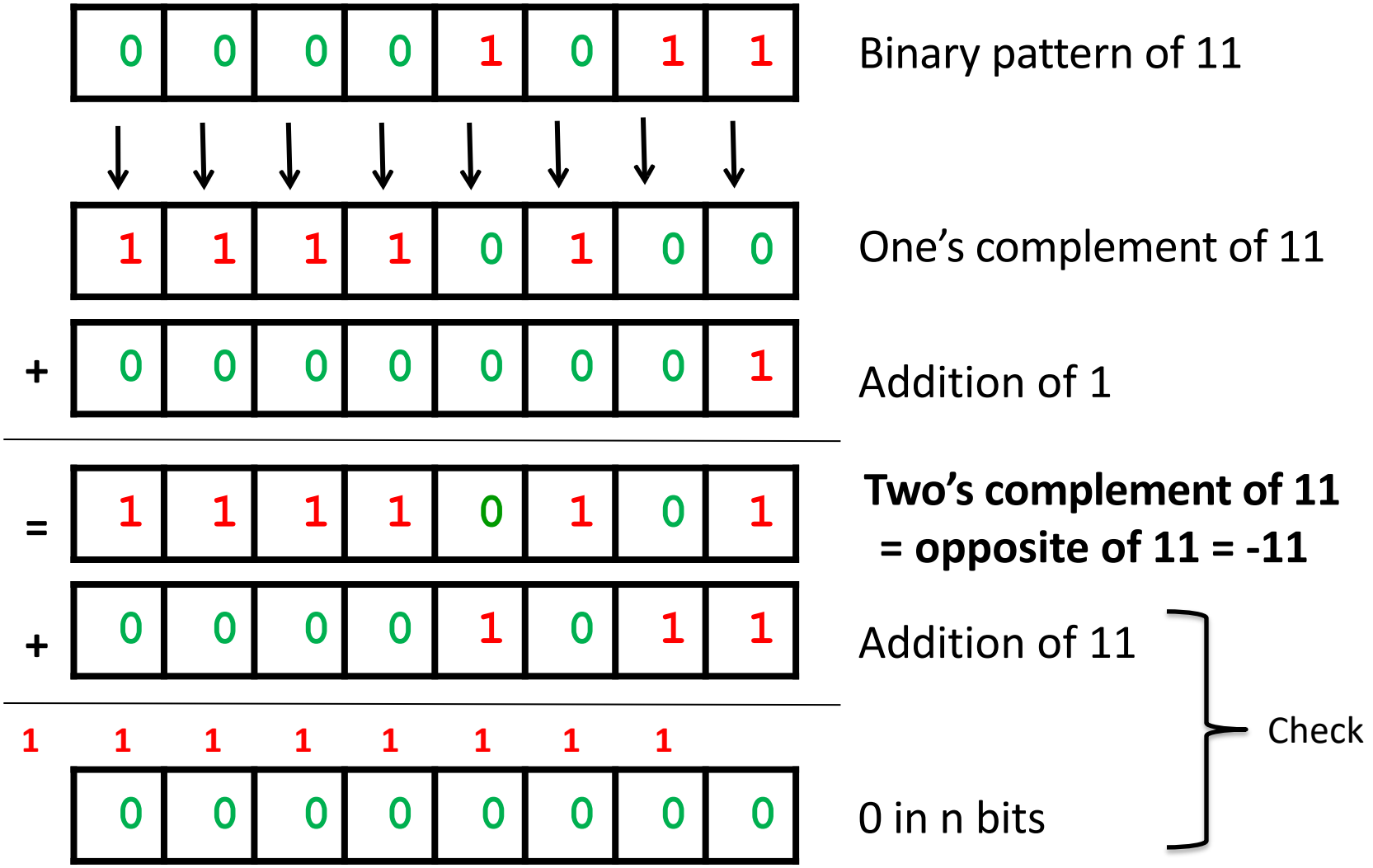
- **Two's Complement = One's Complement + 1**

Example: One's Complement

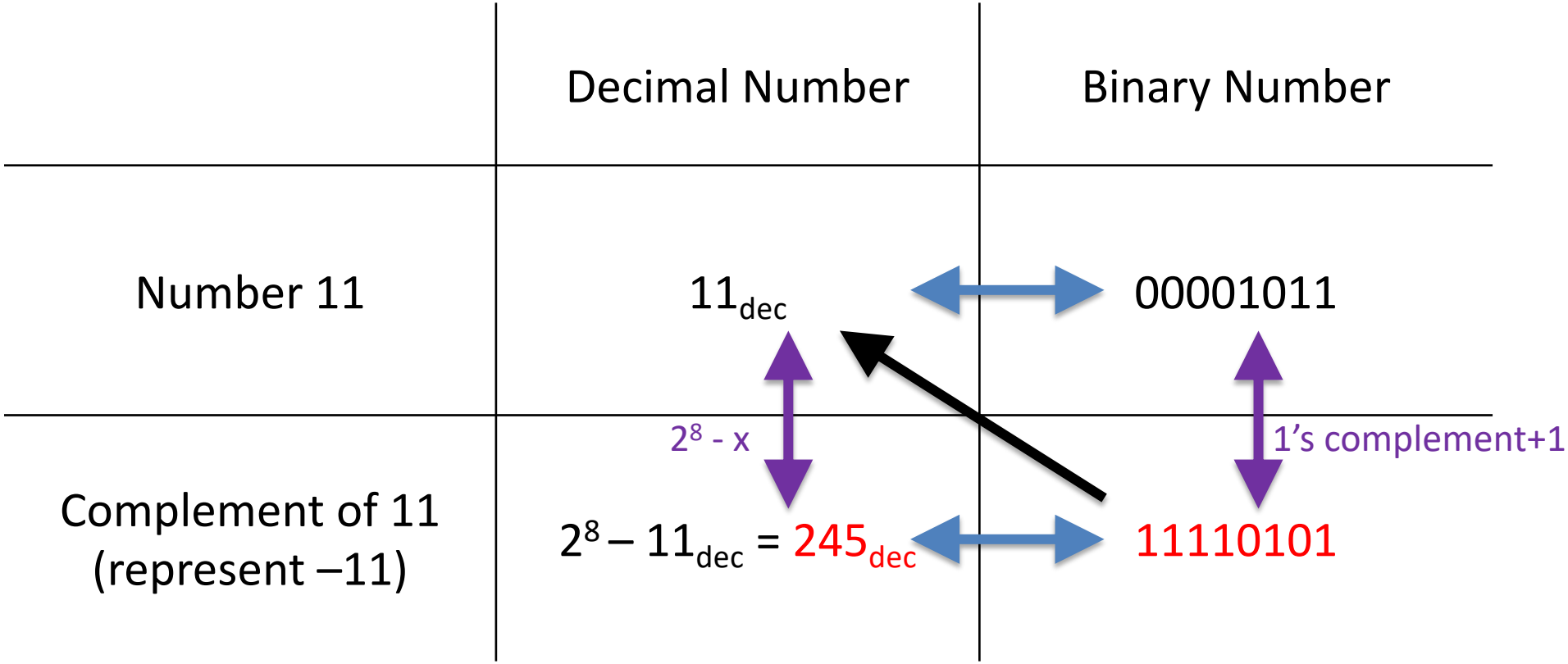
- One's complement of the number 11_{dec} with 8 bits



Example: Two's Complement (+ Check)



Example: Two's Complement

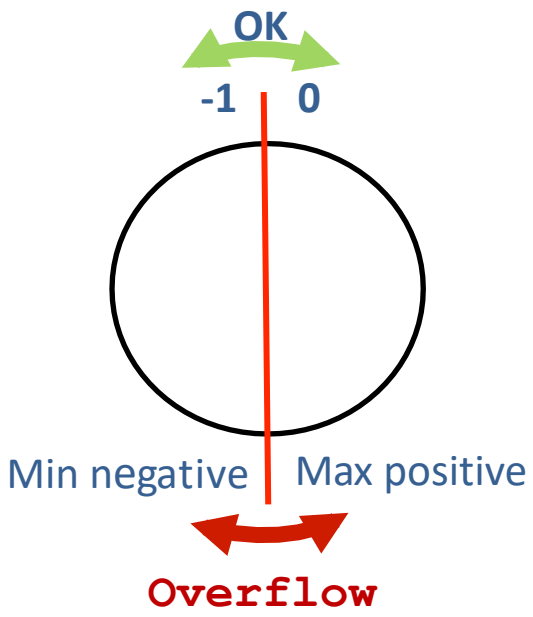


Black arrow: $- 2^7 + 2^6 + 2^5 + 2^4 + 2^2 + 2^0 = - 128 + 117_{dec} = -11_{dec}$

Covered Domain with Two's Complement

- MSB (most significant bit) = sign

Min positive =	00000…0000	= 0
Max positive =	01111…1111	= $2^{(n-1)}-1$
Min negative =	10000…0000	= $-2^{(n-1)}$
Max negative =	11111…1111	= -1



- **Overflow**: incorrect change of sign bit

Signed Integers: Covered Domain and Overflow

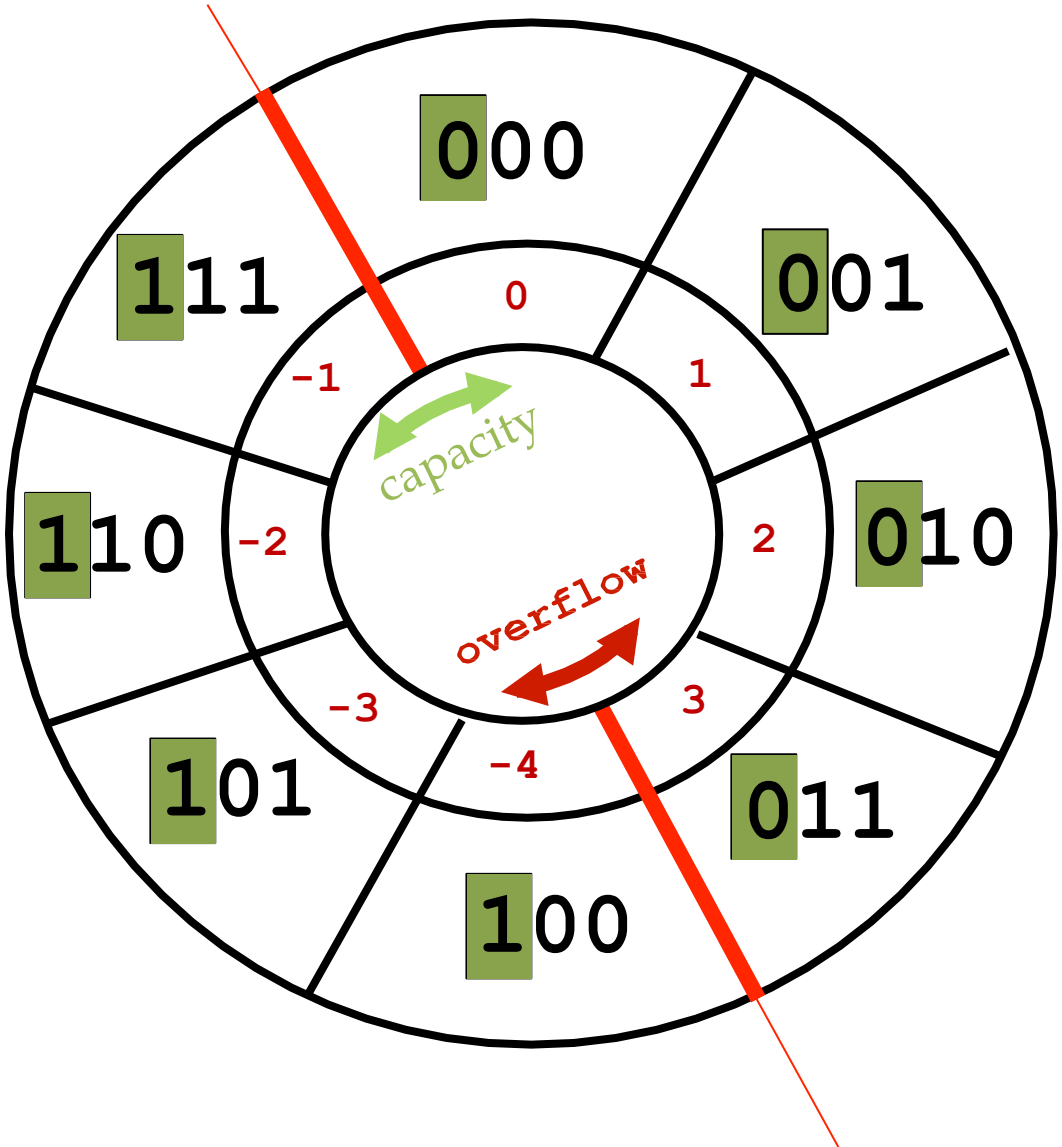
Example:
on 3 bits

Min pos. = **000** = 0

Max pos. = **011** = 3

Min neg. = **100** = -4

Max neg. = **111** = -1



Summary

- We represent a **positive integer** using **positional representation in base 2**.
- We represent a **negative integer** by its **two's complement.**

Which Representation?

- How do you know which representation is used (e.g., in an exam)? Part of the instructions!
- What tells the computer which representation it should use? The **type**!
- Integer Types in C++:
  - C++ uses the **positional representation in base 2** to represent positive integers in the type (**unsigned**) **int**
  - C++ uses the Two's Complement to represent negative integers in the type **int.**

Should You Care About the Covered Domain?

- In C++ integers are represented with 32 bits by default
- What does the following program print?

```
unsigned int i(0);
i = i - 1;
cout << "0 - 1 gives " << i << endl;

int j(2147483647);
j = j + 1;
cout << "2'147'483'647 + 1 gives " << j <<
endl;
// 2^31 - 1 = 2'147'483'647
// 2^32 - 1 = 4'294'967'295
```

```
==================================
Example with non-negative integers
==================================
0 - 1 gives 4294967295
======================
Example with integers
======================
2'147'483'647 + 1 gives -2147483648
```

Agenda

- Representation of the information

- Representation of
  - Natural Numbers (e.g., 2 4 5 6): operations/domain
  - Integers (e.g., -1 -5 4 45698)
  - **Reals (e.g., 3.4 4.756): fixed and floating point**
  - Alphabets and Images

Representation of Real Numbers

- How to we represent real numbers in base 10?
  We use a **decimal point** and **negative powers of 10**,
  e.g., 3.14 is a shortcut for $3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$

- **Same idea for binary**: **negative powers of 2**, e.g.,
  1.011 is a shortcut for $1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$
  (Recall: $2^{-1}=0.5$, $2^{-2}=0.25$, $2^{-3}=0.125$)
  Therefore, 1.011_{bin} represent 1.375_{dec}

Example of Conversion

- How to represent 3.625_{dec} in binary?
  - Step 1: convert 3_{dec} to binary → 11_{bin}
  - Step 2: convert 0.625_{dec} to binary
- **Algorithm**: repetitive multiplication with 2

$$
\begin{array}{lllll}
 & & & b_i & \\
0.625 \cdot 2 = 1.25 & = & 1 & + 0.25 & \Rightarrow 2^{-1} \\
0.25 \cdot 2 = 0.5 & = & 0 & + 0.5 & \Rightarrow 2^{-2} \\
0.5 \cdot 2 = 1 & = & 1 & + 0 & \Rightarrow 2^{-3}
\end{array}
$$

$$0.625_{dec} = 0.101_{bin}$$

$$
\begin{aligned}
& 0.625 \\
&= 2^{-1} \cdot (1 + 0.25) \\
&= 2^{-1} \cdot (1 + 2^{-1} \cdot (0 + 0.5)) \\
&= 2^{-1} \cdot (1 + 2^{-1} \cdot (0 + 2^{-1} \cdot (1 + 0))) \\
&= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\
&= .101
\end{aligned}
$$

$$3.625_{dec} = 11.101_{bin}$$

Another Example

- Represent 0.1_{dec} in binary:

0.1	·	2	=	0.2	=	0	+	0.2
0.2	·	2	=	0.4	=	0	+	0.4
0.4	·	2	=	0.8	=	0	+	0.8
0.8	·	2	=	1.6	=	1	+	0.6
0.6	·	2	=	1.2	=	1	+	0.2
0.2	·	2	=	0.4	=	0	+	0.4
0.4	·	2	=	0.8	=	0	+	0.8
…	…	…	…	…	…	…	…	…

0.1_{dec} = 0.0 0011 0011 0011…

How do we represent 1/3 in decimal? 0.333333….

Consequence of Finite Number of Digits

- We can only use a **finite** number of digits (when we write down a number or on a computer), which leads to a **loss of precision**.
- With n digits we can represent only **2^n numbers precisely** (without loss of precision) but in any interval, e.g., from 0 to 1, there are infinitely many real numbers. So, most of these numbers are represented with an error.

Discretization* Error

- **Absolute error**

$$\delta x = |\, x_{exact} - x_{approx}|$$

- **Relative error**

$$\frac{\delta x}{x} = \frac{|\, x_{exact} - x_{approx}|}{x_{exact}}$$

- The error depends on (i) number of bits, (ii) the size of the interval, and (iii) the chosen representation.

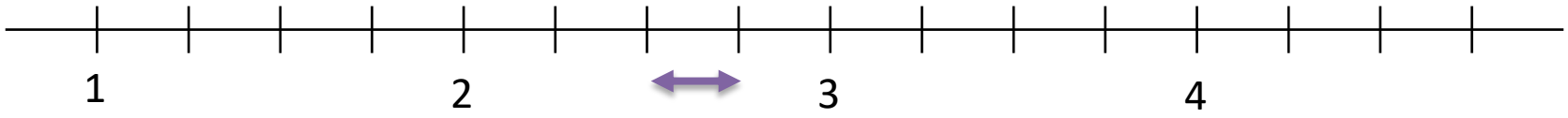
*aka rounding, approximation, quantization error

Example Discretization Error

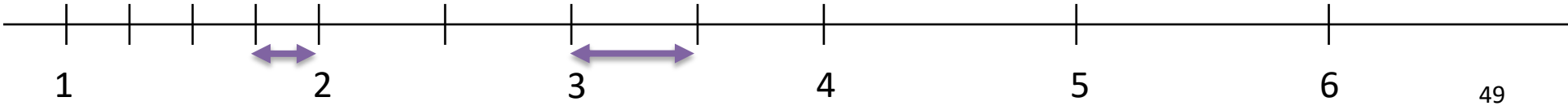
- Assume the number 2.99999 is represented by the number 2.
  - Absolut error: $|2.99999 - 2| = 0.99999 \approx 1$
  - Relative error: $|2.99999 - 2|/2.99999 \approx 1/3 \approx 33\%$
- Assume the number 2000.99999 is represented by the number 2000.
  - Absolut error: $|2000.99999 - 2000| = 0.99999 \approx 1$
  - Relative error: $|2000.99999 - 2000|/2000.99999 \approx 1/2000 \approx 0.005\%$

Representations of Real Numbers

1. **Fixed-point representation**: the fractional point is at a **fixed** position and the distance between any two precisely representable numbers is **constant**.
   - **Absolute error bounded by a constant**
   - **Relative error is not uniform**



2. **Floating-point representation:** the number of bits before and after the fractional point can change between precisely representable numbers, and the distance between two precisely representable numbers can **change** as well.
   - **Absolute error is not uniform**
   - **Relative error bounded by a constant**

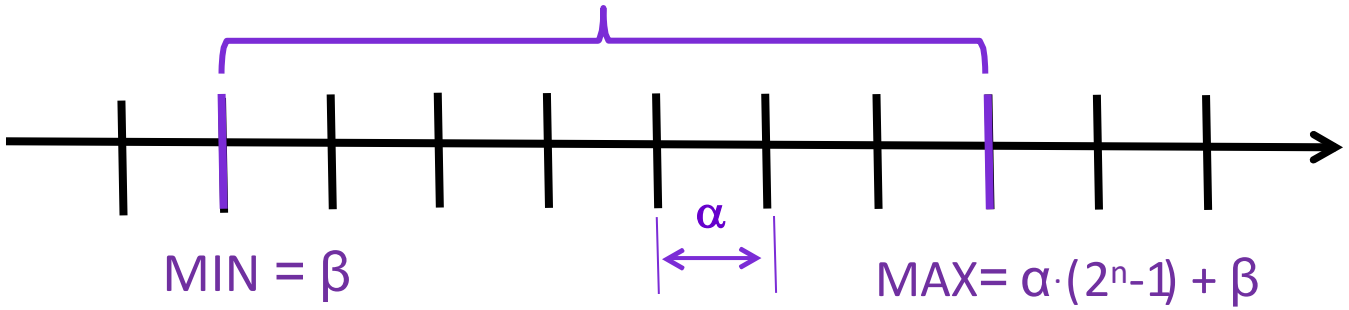


Representative Number

- Numbers that cannot be represented precisely are **truncated**, i.e., bits that do not fit are **cut off.**
- **Example**:
  - Assume we present the numbers from 0 to 3.75 with 4 bits (2 before and 2 after the fractional point).
  - Only 16 numbers can be represented without error: 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2,…, 3.75.
  - The number $0.625_{dec} = 0.101_{bin}$ is then represented by $0.10_{bin} = 0.5_{dec}$ with an absolute error of 0.125_{dec} and a relative error of 0.125/0.625=20%

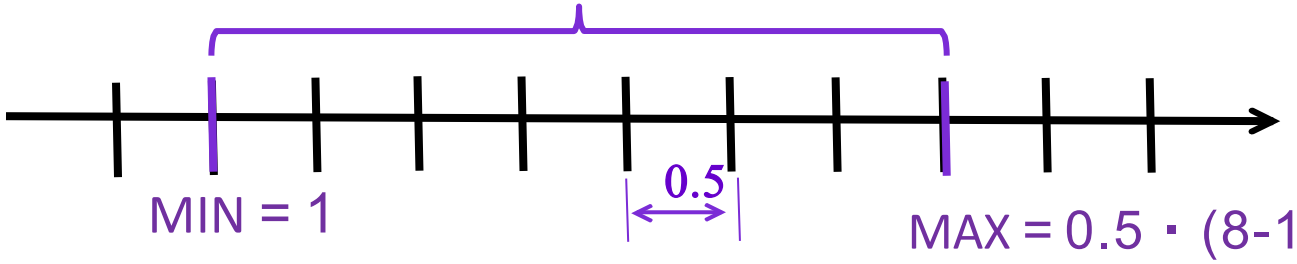
Fixed-Point Representation

- This domain can be put on another scale with **factor α** and shifted by **offset β** to represent decimal numbers.
- For n bits, the 2^n values represented are uniformly spread in the new interval [MIN, MAX] and separated by the quantity α.

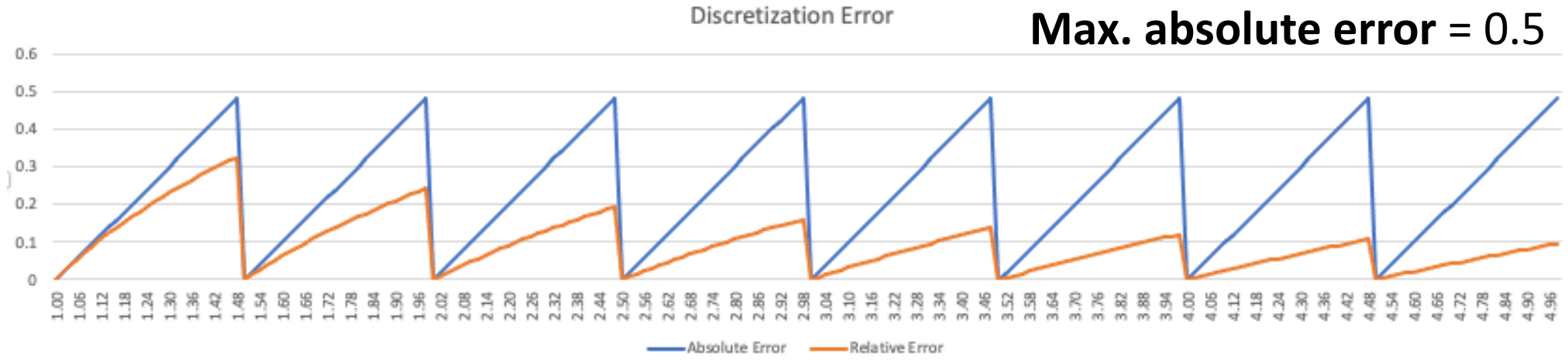


Example: Fixed-Point Representation

- 3 bits to represent decimal numbers in [1,4.5]



Pattern	000	001	010	011	100	101	110	111
In dec.	0	1	2	3	4	5	6	7
Repr.	1	1.5	2.0	2.5	3.0	3.5	4.0	4.5

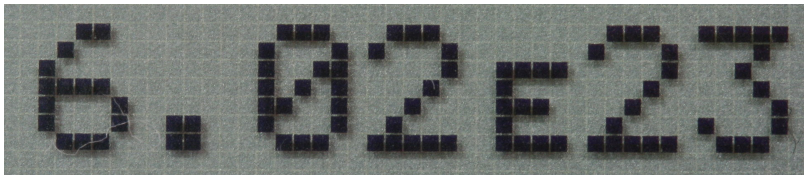


Max. absolute error = 0.5

This relative error distribution is not acceptable for many problems.

Representation with Uniform Relative Error

- **Inspiration**: scientific notation in base 10 with a fixed number of significant digits
  - A small number: $2.4345 \cdot 10^{-10}$
  - A large number: $4.5313 \cdot 10^{8}$



  - **Normalized** number: **1 digit before the fractional point**
    - $356.722 \cdot 10^{0} = 3.56722 \cdot 10^{2}$ (normalized)

- **The relative error is bounded by the number of significant digits.**

Floating-Point Representation

= a scientific notation in base 2

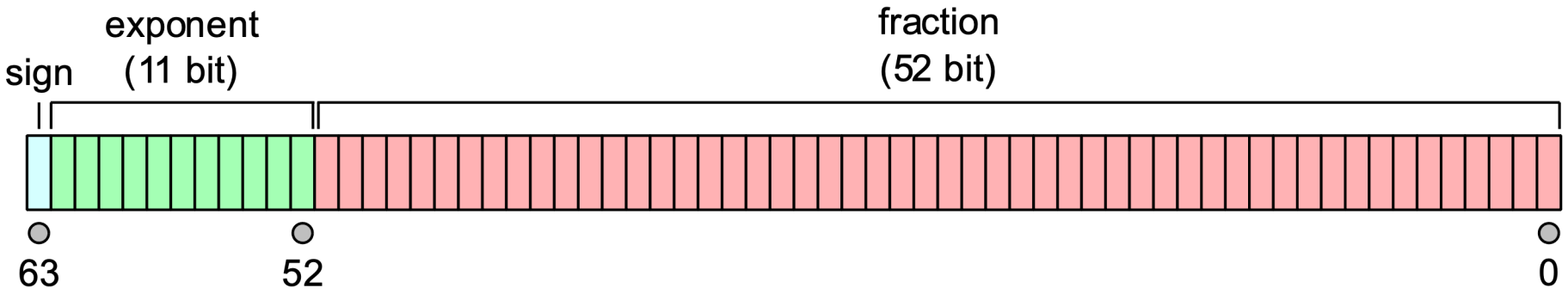
- The floating-point representation in base 2 includes three parts that share the numbers of available bits:
  - the **sign**,
  - the **exponent** of base 2 of the normalized number,
  - the fractional part of the normalized number called the **mantissa**.

Floating-Point Representation

- The **particularity of base 2**: the most significant digit of the normalized number is a constant and always equal to 1. It is, therefore, implicit and does not need to be stored.

$$\boxed{-}\ 1.\boxed{01010100}\ \cdot\ 2^{\boxed{010101}}$$

sign — mantissa — exponent



[https://en.wikipedia.org/wiki/Double-precision_floating-point_format]

Example of Floating-Point Representation

- Let's assume we need to present 20.85_{10} with 8 bits (5 for the mantissa, 3 for the exponent, ignoring the sign)
- Let's convert 20.85 into binary first:
  - 20_{10} in binary is 10100
  - 0.85_{10} in binary is 0.1101100111..
  - 20.85_{10} in binary is 10100.1101100111…
- Normalize the number (1 digit before frac. point):
  $10100.1101100111\ldots = 1.01001101100111\ldots \cdot 2^{4}$
- Convert exponent: 4 in binary is 100.
- Fit into available bits: $1.01001 \cdot 2^{100}$

mantissa

exponent

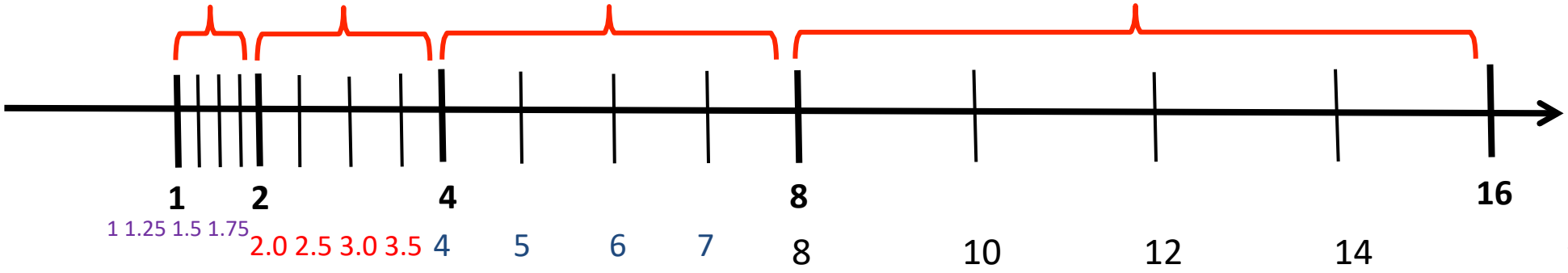
Binary pattern: 10001001 10100.1 = $2^4 + 2^2 + 2^{-1} = 20.5$

Example with Unsigned Exponent

- Example with 2 bits for the **unsigned** exponent and 2 bits for the mantissa. We have the form $1.dd \cdot 2^{dd}$
  The relative error is at most $2^{-2} = ¼$.

Interval	[1-2)	[2-4)	[4-8)	[8-16)
Exp.	00	01	10	11
Mantissa	00	00	00	00
	01	01	01	01
	10	10	10	10
	11	11	11	11

Exponent ranges from 0 to 3

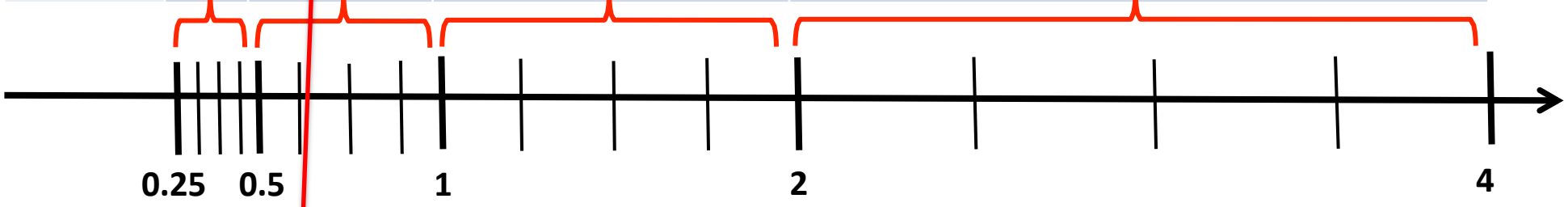


Example with Signed Exponent

- Example with 2 bits for the **signed** exponent and 2 bits for the mantissa. We have the form $1.dd \cdot 2^{dd}$
  The relative error is at most $2^{-2} = ¼$.

Interval	[0.25 -0.5)	[0.5-1)	[1-2)	[2-4)
Exp.	10	11	00	01
Mantissa	00	00	00	00
	01	01	01	01
	10	10	10	10
	11	11	11	11

Exponent ranges from -2 to 1



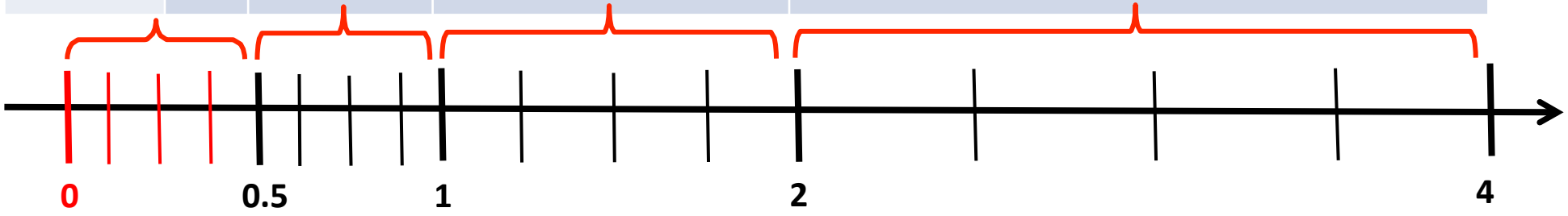
$Bit\ pattern: 1101 \Rightarrow 1.01 \cdot 2^{11} = 1.01 \cdot 2^{-1} = 0.101 = 0.625_{10}$

Representing 0 in Floating-Point Representation

- To include 0 we treat the **lowest power** of 2 (denoted by P) as a special case and use it to cover the interval [0, 2^p].

Interval	[0-0.5)	[0.5-1)	[1-2)	[2-4)
Exp.	10	11	00	01
Mantissa	00	00	00	00
	01	01	01	01
	10	10	10	10
	11	11	11	11

Exponent ranges from -2 to 1



Summary

- Fixed-Point Representation:
  - Integer representation scaled to the desired range with factor and offset. Done **manually** by user!
  - Distance between any two precisely representable numbers is constant
- Floating-Point Representation
  - **Scientific representation in base 2** (with sign, exponent and mantissa)
  - The relative error is bound by 2^{-n}, where n is the size of the mantissa
  - The number 0 requires special treatment
  - **Types in C++: float (32 bits) and double (64 bits)**

Why do I need to know that?

- Recall the **discretization (aka rounding) error is the rule**, i.e., most real numbers are represented with an error.
- This can influence your computation.

A Problematic Example

- Quadratic equation $a \cdot x^2 + b \cdot x + c = 0$ with the decimal values a=0.25, b=0.1, c=0.01
- Discriminant $\Delta = b^2 - 4ac = 0.1^2 - 4 \cdot 0.25 \cdot 0.1 = 0$ but if a, b, and c are represented as floating-point numbers with 64 bits, then $\Delta = 1.73 \cdot 10^{-18}$

```
double a = 0.1 * 0.1; //0.01
double b = 4 * 0.25 * 0.01;// 0.01
if ( a == b) { cout << "a is equal to b" << endl;
} else {       cout << "a is not equal to b" << endl; }

./float_equal
a is not equal to b
```

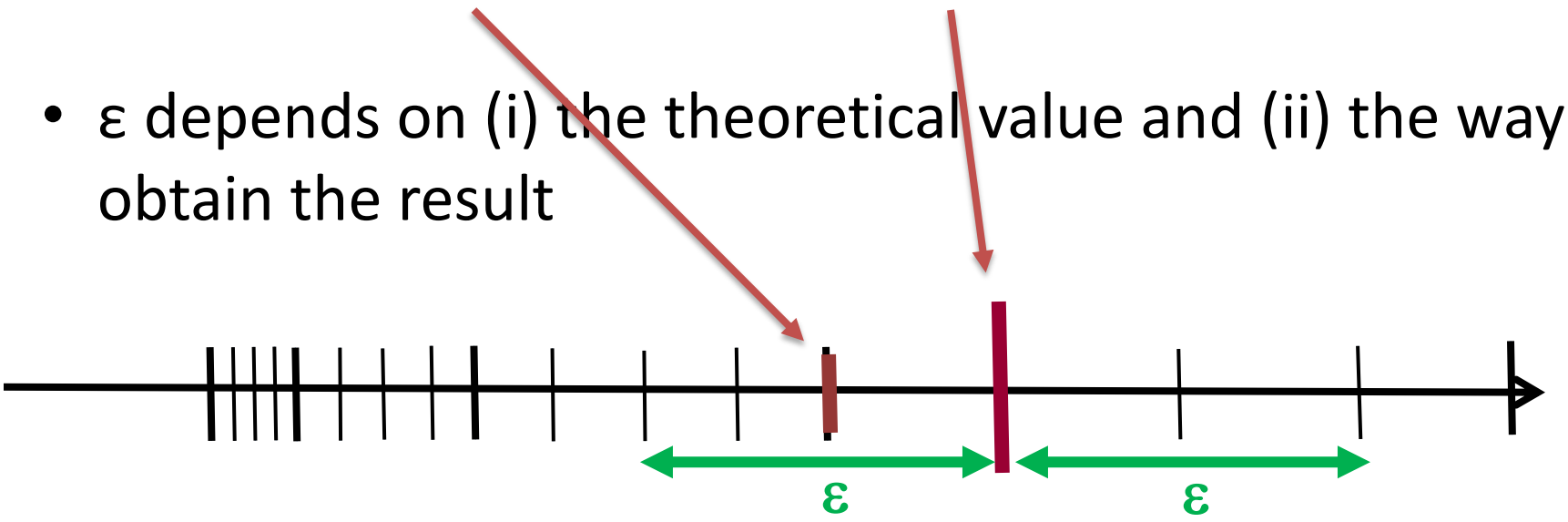
- Why?
  - 4 and 0.25 are exactly represented, their product is 1
  - On the contrary, decimals 0.1 and 0.01 are approximated
- Consequence: $4 \cdot 0.25 \cdot 0.01 \neq 0.1 \cdot 0.1$

Consequence of Rounding Errors

- The equality test must be redefined to allow a tolerance ε around the theoretical value.
- Two values are equal if their distance is smaller than ε

$$| \text{result} - \text{theoretical\_value} | < \varepsilon$$

  - ε depends on (i) the theoretical value and (ii) the way to obtain the result



Consequence of Rounding Errors

- The result is different according to the order of the operations. The **addition is no longer associative**:
- There are values a, b, c such that (a + b) + c ≠ a + (b +c)
- Example with a standard on 64 bits having 52 bits of mantissa:

```
double small(pow(2,-53));
double sum1((1 +  small) + small);
double sum2( 1 + (small  + small));
if (sum1 == sum2) { cout << "sum1 == sum2" << endl;
} else {            cout << "sum1 != sum2" << endl; }
```

```
./double_sum
sum1 != sum2
```

$(1 + 2^{-53}) + 2^{-53}$	→	$1 + 2^{-53}$	→ 1
$1 + (2^{-53} + 2^{-53})$	→	$1 + 2^{-52}$	which is representable

- **Good practice**: first add the **small numbers between them** before adding them to the larger ones

Controlling the Accuracy is Possible

- For a given problem and its algorithmic solution, it is important to ask the following questions:
  - What precision do I need for my results?
  - What is the effect of the algorithm on the precision of the results?
  - What is the maximum available precision on the target machine?
- In the case of insufficient precision, one has to reconsider the algorithmic solution, or adjust the representation in order to obtain the desired precision.

Summary

Shortcut for
$-(2^8 - (2^7 + 2^1 + 2^0))$

A binary pattern **10000011** can have many meanings:

1. Unsigned number: $10000011_{bin} = 2^7 + 2^1 + 2^0 = 130$
2. Signed number: $10000011_{bin} = -(2^7 - 2^1 - 2^0) = -125$
3. Fixed-point number with 6 bits before and 2 after:
   $100000.11_{bin} = 2^5 + 2^{-1} + 2^{-2} = 32.75$
4. Floating-point number with 3 bits for the exponent in 2's complement and 5 for the mantissa :
   $2^{100} \cdot 1.00011 = 0.000100011 = 0.068359375$

-4

Agenda

- Representation of the information

- Representation of
  - Natural Numbers (e.g., 2 4 5 6): operations/domain
  - Integers (e.g., -1 -5 4 45698)
  - Reals (e.g., 3.4 4.756): fix and floating point
  - **Alphabets and Images**

String (char, string), Pictures and Beyond

- Everything is map to numbers using different convention (standards)
- E.g., there are standards (ASCII) to map letters (A, B, C…) to numbers: A=65, B=66, C=67… (7 bit for each letter)
- An image is a grid of pixel (each represented by four numbers with 8 bits)

Letters/Characters

- **ASCII** (American Standard Code for Information Interchange) codes represent letters using 7 bits, value 0-127) http://www.asciitable.com
- **ISO 8859 Latin1** extends ASCII to 8 bits (128-256) to present accented characters lower and upper case of western languages: é è ê à ä ö
- **UNICODE** integrates other languages, > 109'000 characters for 93 writings including Chinese.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

A string is a sequence of letters, each encoded with a number.

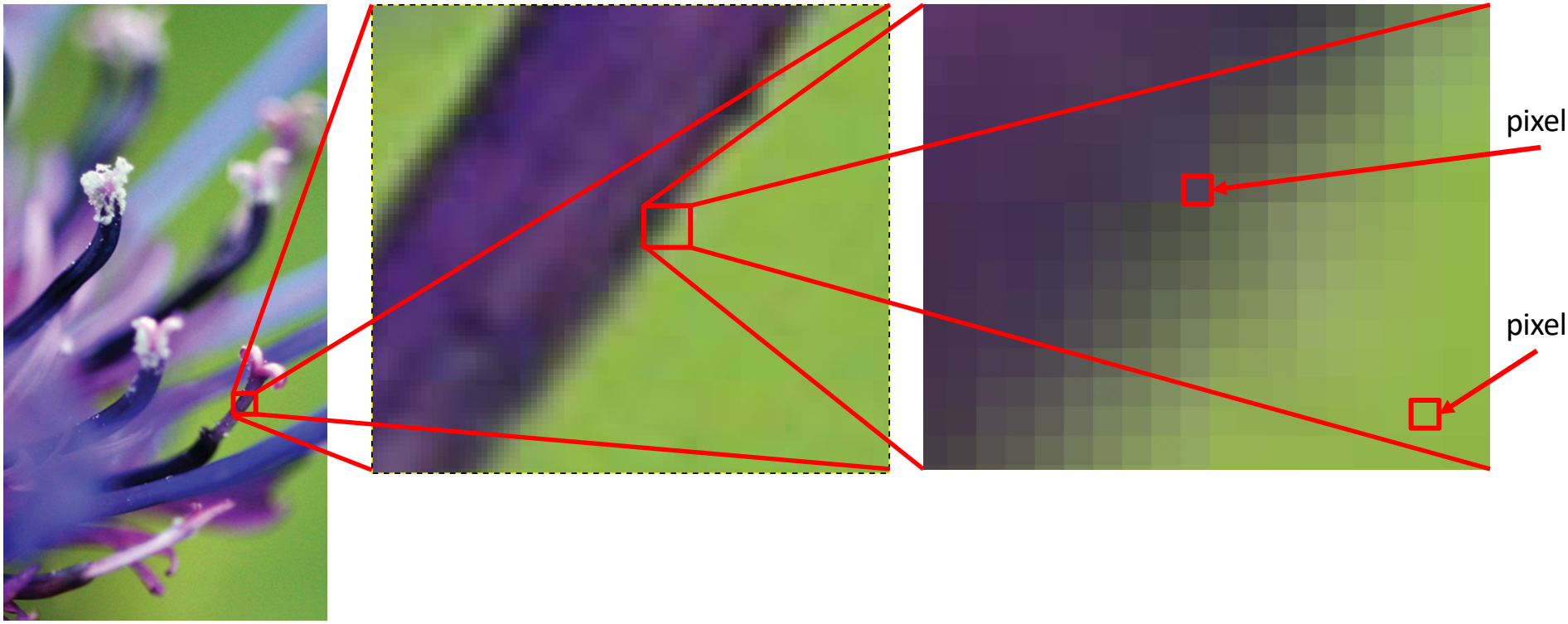
Example in C++

```
#include <iostream>
using namespace std;

int main() {
        int  integer(65);
        char character(65);
        cout << integer << endl;
        cout << character << endl;
}
```

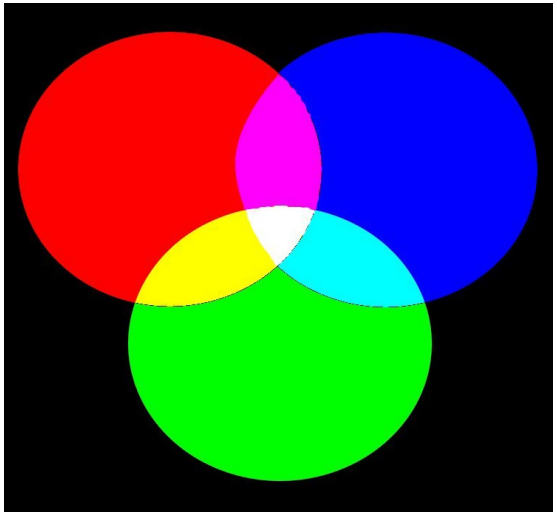
```
65
A
```

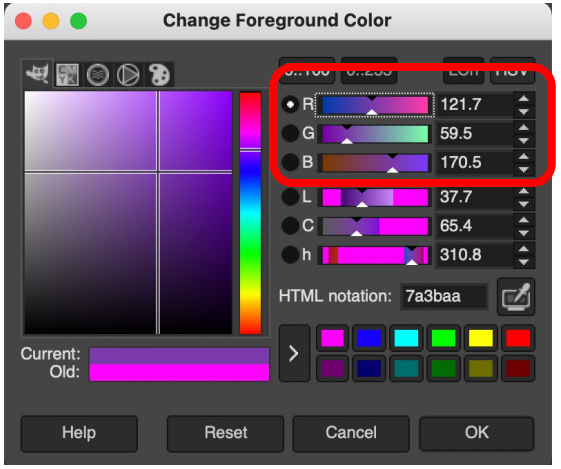
Images



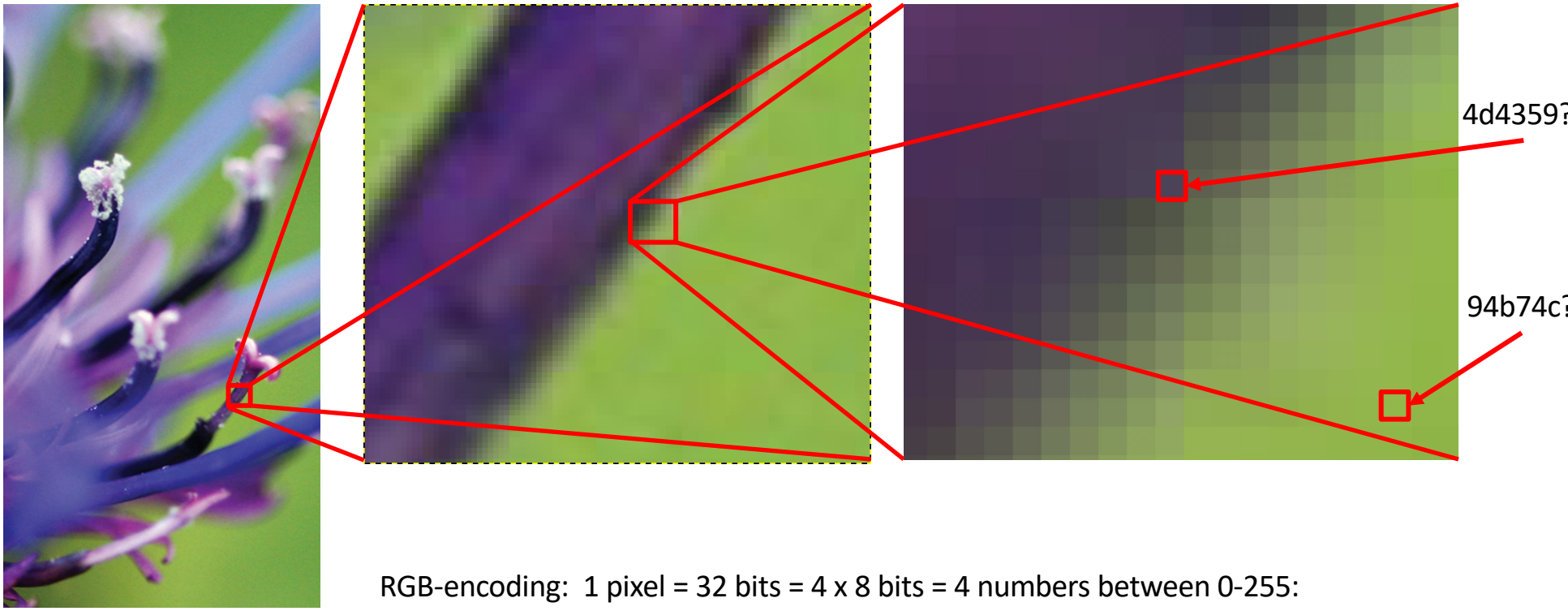
Images

- Each pixel stores the intensity of three primary components whose combination creates a space of colors.
- RGB encoding (Red, Green, Blue)
  - Additive synthesis of the colors:
  - Black=(0,0,0)
  - Red=(255,0,0)
  - Green=(0,255,0)
  - Blue=(0,0,255)
  - White=(255,255,255)
  - Gray levels when three components are equal
- Sometimes completed by a 4th component called alpha (transparency) for graphical applications.

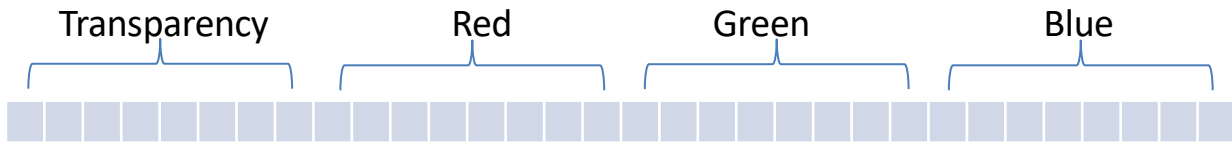




Images



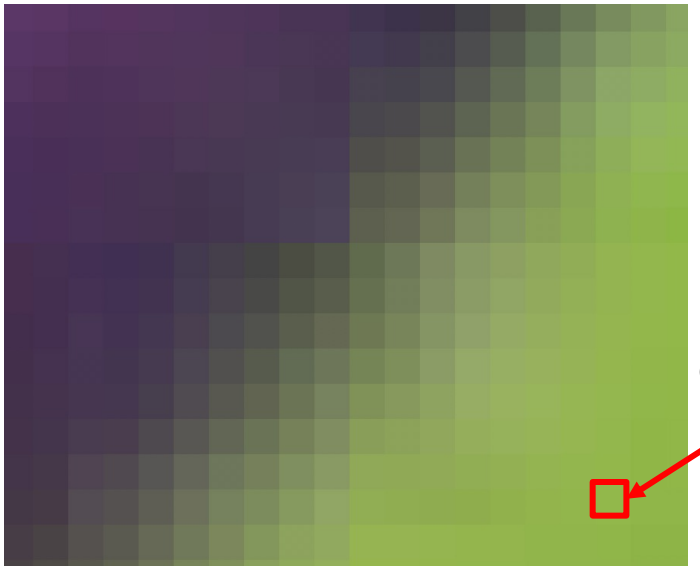
RGB-encoding: 1 pixel = 32 bits = 4 x 8 bits = 4 numbers between 0-255:



Hexadecimal Numbers

- Numbers in base 16 (2^4) = 4 bits

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
..	..	..
9	9	1001
10	a	1010
11	b	1011
12	c	1100
13	d	1101
14	e	1110
15	f	1111
16	10	10000



94b74c =
Red: 94 = 1001_0100 = 148
Green: b7 = 1011_0111 = 183
Blue: 4c = 0100_1100 = 76

Summary

- We have seen representation of
  - **natural numbers** (`unsigned int`),
  - **integers** (`int`),
  - **fractional numbers** (`float` or `double`),
  - **characters** (`char`), and
  - **images** (using discrete sampling).
- A representation is a **human convention** of interpretation of a set of signs. Its power is directly connected to the number of people who share it, hence the importance of standards (e.g., code ASCII, UTF).

Question 1

- Assume we present **unsigned** integers with four bits. What is the result of 5+14?
  - What range can we represent with four bits?
  - Does 19 fit into this range?

A. 19
B. 16
C. 3
D. -16

Question 2

- What is the two's complement representation (in binary) of the number 19 using 8 bits?
  - Recall definition: $2^n - x$
  - Alternative computation?

A. 00010010
B. 00010011
C. 11101100
D. 11101101

Question 3

- Assume we present signed integers with four bits (including the sign). Which of the following computations are possible results in this system? (Multiple answers are possible)

A. 4 + 3 = 7
B. 4 + 4 = 0
C. 4 + 4 = -8
D. 4 - 5 = -1
E. -4 - 5 = -9

Question 4

- Assume the binary pattern 01010010 represents a fractional number in which the first 4 bits represent the (signed) exponent and the last 4 bits represent the mantissa. What is the corresponding decimal number?

A. 15
B. 36
C. 72
D. 82