

# Bits and Bitwise Operations

## Bits, Numerals Systems and Bitwise Operations



SoftUni Team  
Technical Trainers



x	0	0	1	1
y	0	1	0	1
<hr/>				
x & y	0	0	0	1

x	0	0	1	1
y	0	1	0	1
<hr/>				
x   y	0	1	1	1

x	0	0	1	1
y	0	1	0	1
<hr/>				
x ^ y	0	1	1	0



SoftUni



Software University

<https://softuni.bg>

[sli.do](https://sli.do)

**#fund-common**

1. What is a Bit, Byte, KB, MB?
2. Numerals Systems
  - Decimal, Binary, Hexadecimal
  - Conversion between Numeral Systems
3. Representation of Data in Computer Memory
  - Representing Integers, Real Numbers and Text
4. Bitwise Operations:  $\&$ ,  $|$ ,  $\wedge$ ,  $\sim$ 
  - Reading / Writing Bits from Integers





**Bits**

What is a Bit?



- **Bit** == the smallest **unit of data used in computing**
  - Takes only one of **two values**: either a **0** or **1**
- **1 bit** can store anything with **two separate states**
  - Logical values (true / false)
  - Algebraic signs (+ / -)
  - Activation states (on / off)
- Bits are organized in computer memory in sequences of **8 bits**, called **bytes** (octets)

# Bit, Byte, KB, MB, GB, TB, PB



- **Bit** – single **0** or **1**, representing a bit of data
- **Byte** (octet) == **8 bits** == the smallest addressable unit in the computer memory
- **KB** (kilobyte) == **1024 bytes** (sometimes 1000 bytes)
- **MB** (megabyte) == **1024 KB** == **1048576 bytes**
- **GB** (gigabyte) == **1024 MB** == **1073741824 bytes**
- **TB** (terabyte) == **1024 GB** == **1099511627776 bytes**
- **PB** (petabyte) == **1024 TB** == **1125899906842624 bytes**



5

101<sub>b</sub>


0x8

# Numerals Systems

Decimal, Binary and Hexadecimal

# Numeral Systems

- **Numeral system** == system for **representing numbers** in written form using sequence of **digits**
- **Positional numeral systems** == the value of each digit depends on its position
  - These numeral systems have a **base** (e.g., 2, 10, 16)



Decimal (base = 10)	Binary (base = 2)	Hexadecimal (base = 16)
30	111110	1E
45	101101	2D
60	111100	3C



# Decimal Numbers

- Decimal numbers (**base 10**)
  - Represented using 10 digits:
    - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
  - Each position represents a **power of 10**


$$\begin{aligned} 401 &= 4 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = \\ &= 4 \cdot 100 + 0 \cdot 10 + 1 \cdot 1 = \\ &= 400 + 0 + 1 = 401 \end{aligned}$$

- A decimal number  $d_{n-1}d_{n-2} \dots d_1d_0 =$   
 $d_0 \cdot 10^0 + d_1 \cdot 10^1 + d_2 \cdot 10^2 + \dots + d_{n-1} \cdot 10^{n-1}$

# Binary Numbers

- The **binary system** is used in computer systems
- Binary numbers (**base 2**)
  - Represented by **sequence of 0 or 1**

$$5 == 101_b$$

- Each position represents a **power of 2**

$$101_b = 1*2^2 + 0*2^1 + 1*2^0 = 4 + 0 + 1 = 5$$

$$1010_b = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8 + 0 + 2 + 0 = 10$$



# Binary and Decimal Conversion

## ■ Binary to decimal

- Multiply each digit to its magnitude (power of 2)

$$\begin{aligned} 1011_b &= 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = \\ &= 1*8 + 0*4 + 1*2 + 1*1 = \\ &= 8 + 0 + 2 + 1 = \\ &= 11 \end{aligned}$$

## ■ Decimal to binary

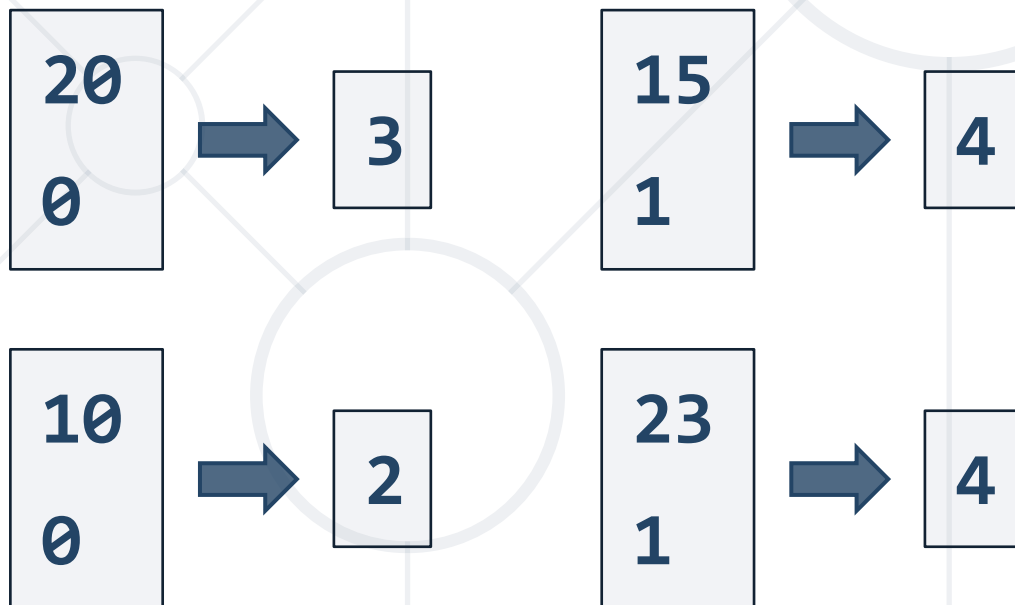
- Divide to the base (2) until 0 is reached and take the remainders in reversed order

$$\begin{aligned} 11 / 2 &= 5 \text{ (1) } // \text{ last digit} \\ 5 / 2 &= 2 \text{ (1) } // \text{ previous digit} \\ 2 / 2 &= 1 \text{ (0) } // \text{ previous digit} \\ 1 / 2 &= 0 \text{ (1) } // \text{ fist digit} \\ \text{Result: } &1011 \end{aligned}$$



# Problem: Binary Digits Count

- You are given a positive integer **n** and a binary digit **b** (0 or 1)
- Write a program that finds the count of **b** digits in the binary representation of **n**



0 1

# Solution: Binary Digits Count

1. **Read the input** from the user: **n** and **b**
  2. **Convert the input to binary** system  
(collect the remainders of division by 2)
  3. **Count the digits b** in the remainders of **n**
  4. Print the **count**
- Another solution is to use **bitwise operations** (think how later)

# Hexadecimal Numbers

- Hexadecimal numbers (**base 16**)
  - Represented using **16 literals** (hex digits)
    - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F**
- Usually **prefixed with 0x** in computer science
- Each position represents a **power of 16**


$$\begin{aligned} 0x\text{B7F6} &= \text{B} \cdot 16^3 + \text{7} \cdot 16^2 + \text{F} \cdot 16^1 + \text{6} \cdot 16^0 = \\ &= \text{11} \cdot 4096 + \text{7} \cdot 256 + \text{15} \cdot 16 + \text{6} \cdot 1 = \\ &= 45056 + 1792 + 240 + 6 = 47094 \end{aligned}$$

# Hex ↔ Decimal Conversions

## ■ Hexadecimal to decimal

- Multiply each digit to its weight (power of 16)

$$\begin{aligned} 0x1F4 &= 1 \cdot 16^2 + 15 \cdot 16^1 + 4 \cdot 16^0 = \\ &= 1 \cdot 256 + 15 \cdot 16 + 4 \cdot 1 = \\ &= 256 + 240 + 4 = \\ &= 500 \end{aligned}$$

## ■ Decimal to hexadecimal

- Divide by 16 and take the remainders in reversed order

$$\begin{aligned} 500 / 16 &= 31 \text{ (4)} \\ 31 / 16 &= 1 \text{ (F)} \\ 1 / 16 &= 0 \text{ (1)} \\ \text{Result: } &0x1F4 \end{aligned}$$



# Hex $\leftrightarrow$ Binary Conversions

- The conversion from **binary** to **hexadecimal** (and back) is straightforward
  - Each hex digit corresponds to a **sequence of 4 binary digits**

A2E3F = 1010 0010 1110 0011 1111

A = 1010

2 = 0010

E = 1110

3 = 0011

F = 1111

1010 0010 1110 0011 1111 = A2E3F

$1010_b = 10_{dec} = A_{hex}$

$0010_b = 2_{dec} = 2_{hex}$

$1110_b = 14_{dec} = E_{hex}$

$0011_b = 3_{dec} = 3_{hex}$

$1111_b = 15_{dec} = F_{hex}$





# **Representation of Data**

Integers, Floating-Point Numbers and Text

# Representing Integers in Memory

- **Integer numbers** are sequences of bits
- Can be **signed** (in most cases) or **unsigned**
  - The **sign** == the **Most Significant Bit (MSB)**
  - Leading **0** → **positive number**
  - Leading **1** → **negative number**
- Example (8-bit signed integers)

**0**XXXXXXX<sub>b</sub> > 0 // 00010010<sub>b</sub> = 18

**0**0000000<sub>b</sub> = 0

**1**XXXXXXX<sub>b</sub> < 0 // 10010010<sub>b</sub> = -110



# Representation of Signed Integers

- **Positive 8-bit** numbers have the format **0XXXXXXX**
  - The value is the decimal value of their last **7 bits (XXXXXXX)**
- **Negative 8-bit** numbers have the format **1YYYYYYYY**
  - The value is **-128** ( **$-2^7$** ) + the decimal value of **YYYYYYYY**

$$\begin{aligned} 10010010_b &= -2^7 + 0010010_b = \\ &= -128 + 18 = \\ &= -110 \end{aligned}$$

**$-2^7$**

# Largest and Smallest Signed Integers

- The **largest signed 8-bit integer** is

$$127 = (2^7 - 1) = \mathbf{0}1111111_b$$

- The **smallest negative 8-bit integer** is

$$-128 = -(2^7) = \mathbf{1}0000000_b$$

- The **largest signed 32-bit integer** is

$$2147483647 = (2^{31} - 1) = \mathbf{0}111...1111_b$$

- The **smallest negative 32-bit integer** is

$$-2147483648 = -(2^{31}) = \mathbf{1}000...0000_b$$

$$2^7 - 1$$

$$-2^7$$

$$2^{31} - 1$$

$$-2^{31}$$

# Integers and Their Ranges in Programming

Bits	Sign	Range	Data Types
8-bit	signed	-128 ... 127 ( $-2^7 \dots 2^7-1$ )	<b>sbyte</b> in C#, <b>byte</b> in Java
8-bit	unsigned	0 ... 255 ( $2^0 \dots 2^8-1$ )	<b>byte</b> in C#
16-bit	signed	-32768 ... 32767 ( $-2^{15} \dots 2^{15}-1$ )	<b>short</b> in C#, <b>short</b> in Java
32-bit	signed	-2,147,483,648 ... 2,147,483,647 ( $-2^{31} \dots 2^{31}-1$ )	<b>int</b> in C#, <b>int</b> in Java

# Representing Real Numbers

- Computers use the **floating-point number** format, defined by the **IEEE 754 technical standard**
- The **IEEE-754** standard defines:
  - Arithmetic and exchange formats – representations of the binary and decimal floating-point data
  - Rounding rules for floating-point numbers
  - Operations – arithmetic and other operations
  - Special numbers – such as **infinity** and **NaN**



# Storing Floating-Point Numbers

- Floating-point numbers are stored as sequence of bits:  
**sign bit**, **exponent** and **mantissa**

Play with the IEEE-754 converter online:  
<https://h-schmidt.net/FloatConverter/IEEE754.html>




- Note: **errors in calculations** and **precision** may occur
  - Some numbers (e.g., 0.3) cannot be represented in the above format without rounding (as a sum of negative powers of 2)

# Representing Text

- Computers represent **text characters** as unsigned integer numbers (i.e. as sequence of bits)
  - Letters, digits, punctuation chars, etc.
- The **ASCII** standard represent chars as **8-bit integers**
  - Defines the ASCII code for 127 chars, e.g.

A



Binary	Dec	Hex	Char
0b01000001	65	0x41	A
0b01000010	66	0x42	B
0b00101011	43	0x2B	+



# Representing Unicode Text

- The **Unicode** standard represents 100,000+ text characters as **16-bit integers** (see [unicode.org](https://unicode.org))
- Supports many alphabets, e.g., Latin, Cyrillic, Arabic



Decimal	Hex	Char	Explanation
65	0x0041	A	Latin "A"
1097	0x0449	Щ	Cyrillic letter "Sht"
1576	0x0628	ب	Arabic letter "Beh"
127928	0x1F3B8	🎸	Emoji "Guitar"

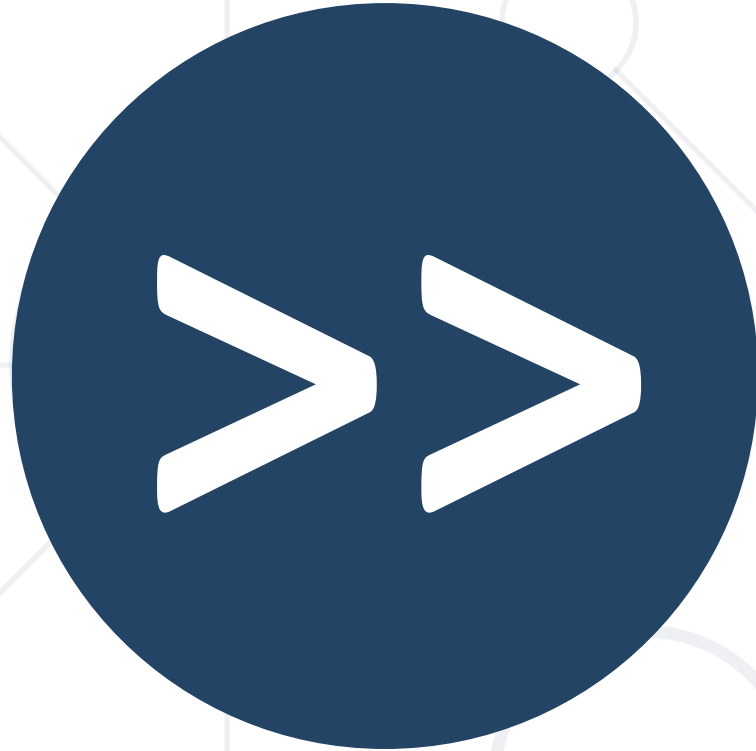
- **UTF-16** uses 2 bytes (16 bits) for each char
- **UTF-8** uses 1, 2, 3 or 4 bytes for each char

- **Strings** represent **text data** in programming
  - **Strings** are **arrays of characters**, typically represented like this

len	0	1	2	3	4
5	H	e	l	l	o

Takes **14 bytes** in memory  
 $4 \text{ bytes (length)} + 5 * 2 \text{ bytes}$

- The string can have its **size as prefix** (used in most languages) or can end with **\0** (null-terminated string – used in C)
- Characters in the string can be
  - **16-bit** (UTF-16) – default in C#, Java, JS, Python
  - **8-bit** (ASCII / windows-1251) – default in C, C++




# Bitwise Operations

Bitwise Operators and Bit Shifts

# Bitwise Operators

- **Bitwise operators** works with the binary representations of the numbers, applying **bit by bit** calculations
- The operator **~** turns all **0** to **1** and all **1** to **0** (like **!** for boolean expressions but bit by bit)
- The operators **|**, **&** and **^** behave like **||**, **&&** and **^** for boolean expressions but bit by bit



Operator					&	&	&	&	^	^	^	^
Operand1	0	0	1	1	0	0	1	1	0	0	1	1
Operand2	0	1	0	1	0	1	0	1	0	1	0	1
Result	0	1	1	1	0	0	0	1	0	1	1	0

# Bitwise Operators – Examples

## ■ Bitwise NOT (~)

5	//	0101
~5	//	1010

## ■ Bitwise OR (|)

5	//	0101
3	//	0011
5   3	//	0111

## ■ Bitwise AND (&)

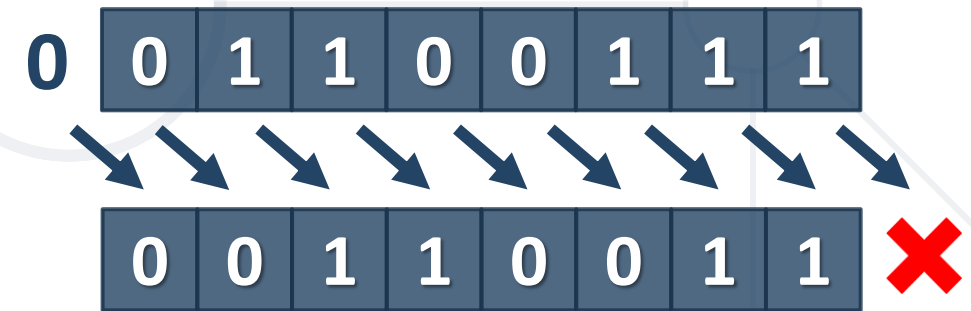
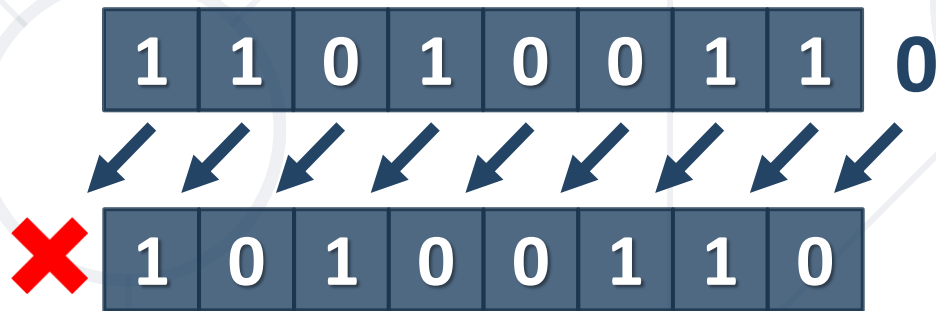
5	//	0101
3	//	0011
5 & 3	//	0001

## ■ Bitwise XOR (^)

5	//	0101
3	//	0011
5 ^ 3	//	0110

# Bit Shifts

- **Bit shifts** are bitwise operations, where
  - Bits are moved (**shifted**) to the **left** or **right**
  - The bits that fall outside the number are **lost** and **replaced by 0**
- **Left** shift (<< operator)
- **Right** shift (>> operator)



# Bitwise Operations: Get the Last Bit

- How to **get the last bit** from a number **n**?
  - The bits are **numbered from 0**, from right to the left
  - The position of the last (rightmost) bit is **0**

```
n = 125    // 01111101
mask = 1    // & 00000001
n & mask    // 00000001 = 1
```



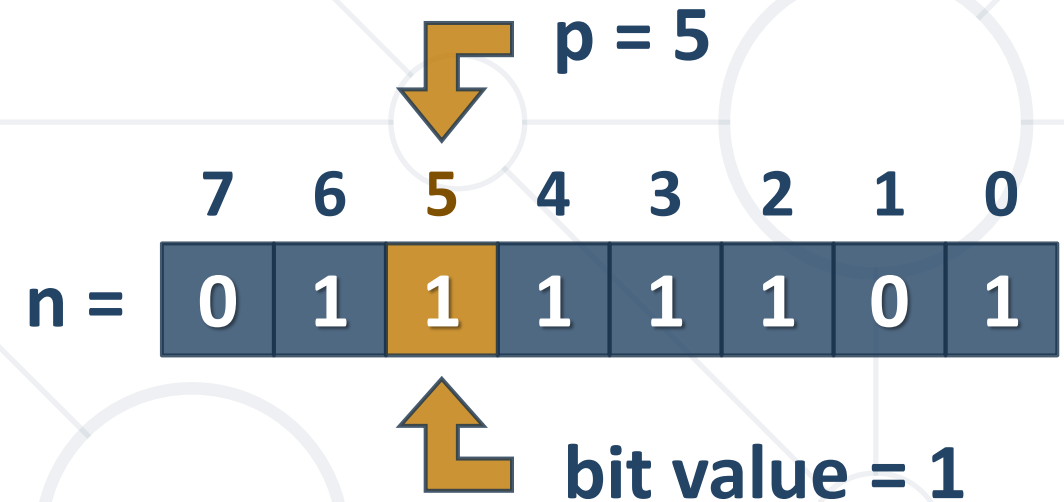
- Last bit – formula:

```
lastBit = n & 1
```

# Bitwise Operations: Get Bit at Position

- How to **get the bit at position p** from a number **n**?

```
n = 125    // 01111101
p = 5      // 5th position
125 >> p   // 00000011 = 3
3 & 1      // 00000001 = 1
```



- Bit at position – formula:

```
bit = (n >> p) & 1
```



# Bitwise Operations: Set Bit at Position

- How to **set the bit** at given position **p** to **0** or **1**?

- Clear** a bit (0) at position **p**

```
p = 5           // 5th position
n = 125         // 01111101
mask = ~(1 << p) // 11011111
result = n & mask // 01011101
```

- Set** a bit (1) at position **p**

```
p = 5           // 5th position
n = 125         // 01111101
mask = 1 << p   // 00100000
result = n | mask // 01111101
```

- Assign a bit **b** (0 or 1) at position **p** – formula:

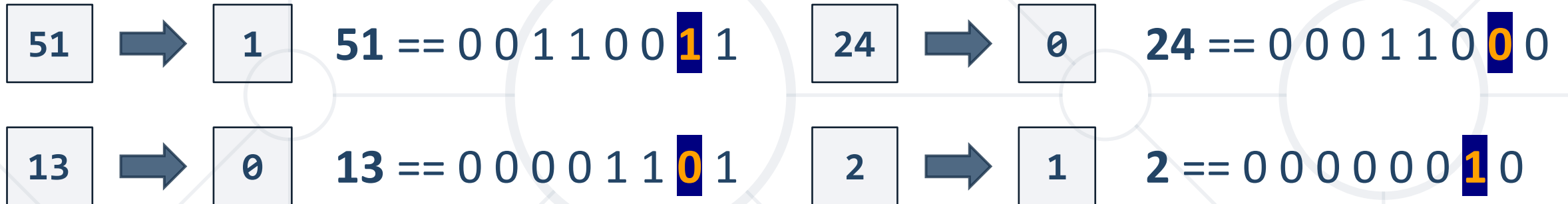
```
n = n & ~(1 << p) | (b << p)
```

# Why We Need Bitwise Operations?

- **Networking protocols**
  - Many devices communicate using bit-level protocols
  - e.g., the SYN flag in the **TCP protocol** header is the bit #1 from the 14<sup>th</sup> byte in the TCP packets
    - Web browsers use bitwise operations to connect to a Web site
- Many **binary file formats** use bits to save space
  - e.g., PNG images use 3 bits to specify the color format used
- **Data compression** replaces byte sequences with bit sequences
  - e.g., the DEFLATE algorithm in **ZIP files**

# Problem: Bit #1 (the Bit Before the Last)

- Write a program that prints the **bit** at **position 1** of an **integer**



- Solution:

```
p = 1           // 1st position
n = 51          // 00110011
n = n >> p      // 00011001 = 25
n & 1           // 1
```

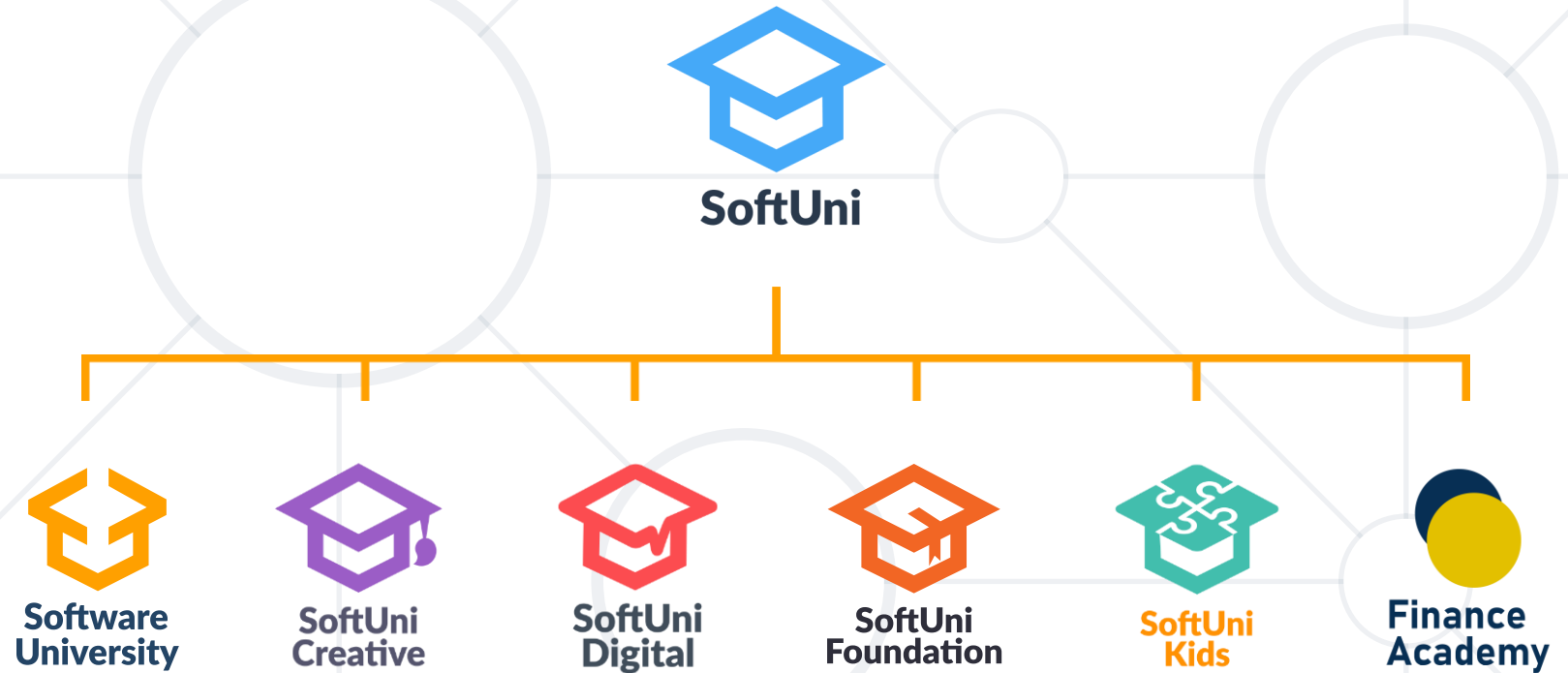


**Live Exercises**

- Computers store data using **bits**
  - Signed **integers** (leftmost bit == sign)
  - **IEEE-754** – floating point numbers
  - **Text** is stored using ASCII / Unicode / other
- **Binary** and **hexadecimal numeral systems** play a key role in computing
- Developers manipulate **bits** in integers using **bitwise operators** and **bit masks**



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)





- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

