Data Representation in Computers

- Values (text, video, audio) – represented as numbers
- Numbers – expressed in ***numeral systems***
- Numeral system types
    - ***Non-positional systems*** (e.g. Roman numerals) – there is no straightforward relationship between the numeral position and its impact on the final value of a number, because it depends on the surrounding numerals; e.g. in the number XXIX the I means "subtract 1 from the following numeral", whilst in the XXII "add 1 to the whole number".
    - ***Positional systems*** (e.g. Babylonian numerals or Hindu-Arabic numerals) – the position of a numeral has straightforward impact on the final value (3 has the same effect in a number 40234 and 139).

Positional Systems

- Positional systems are based on several digits, which multiply different powers of the radix

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_2 b^2 + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + a_{-2} b^{-2} \ldots$$

b is the radix/base – e.g. 10 in the decimal system or 16 in hexadecimal, a_i are digits.
If a system uses *b* radix, there are used digits from *0* to *b-1*.
For example the number 2858_{10} (i.e. in the decimal system) is

$$2 \times 10^3 + 8 \times 10^2 + 5 \times 10^1 + 8 \times 10^0$$

Systems Typical of Computers

Binary System

Because of technical limitations computers use the simplest numeral system – ***binary system*** (or base-2 number system) - straightforward implementation in electronic systems; there are required two states only – ON (1) and OFF (0) – usually two voltage levels.

- Digits: 0 and 1 – **bits**
    - The leftmost one - **the most significant bit (MSB)**
    - The rightmost one – **the least significant bit (LSB)**
- Base: 2
- Example of a binary value: 11001, which is

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

- Significant drawback – **verbosity** (e.g. 260_{10} takes 3 numerals, while its binary form 100000100_2 takes 9 numerals)

Hexadecimal

Base: 16

Digits: 0 – 9, A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

Hexa digits in the binary system:

- 0 = 0000
- 1 = 0001
- 2 = 0010
- 3 = 0011
- 4 = 0100
- 5 = 0101
- 6 = 0110
- 7 = 0111
- 8 = 1000
- 9 = 1001
- 10 = 1010
- 11 = 1011
- 12 = 1100
- 13 = 1101
- 14 = 1110
- 15 = 1111

It is often used to represent content, which is in fact binary; advantages:

- more compact than the binary system – one hexa digit covers a quadruple of bits
- simple conversion from/to binary

Examples of use

- memory content – each byte can be represented by a hexadecimal number with two digits
- color codes in RGB, RGBA – each color channel for a pixel is in one byte – two hexadecimal digits for each channel – six hexadecimal digits for one pixel (in RGB)

1	0	1	1	1	1	0	1	0	1	0	1	0	0	0	1	1	1	1	1	1	0	0	1

B	D	5	1	F	9

- parameters in URL addresses: http%3A%2F%2Fwww%2Egoogle%2Ecom – the text introduced by % represents a hexadecimal value of the character in the ASCII table (e.g. %2F => $2F_{16} = 47_{10}$ = backslash ‘ \ ’).
- character according to the UNICODE standard in XML/XHTML, e.g. &#00e4; means ‘the symbol number $00e4_{16} = 228_{10}$ = ä’.
- IPv6 addresses

Octal system

- base: 8
- digits: 0 – 7
- similar in use to hexadecimal – when triplets of bits are represented

Examples of use:

- file/folder privileges in Linux/Unix – 001_2 means 'execute', 010_2 means 'write', 100_2 means 'read' – e.g. 101_2 means 'allowed to read and execute, but not to write' – in octal 5.

Conversions between Systems

Decimal To Binary

1. Calculate N *mod* 2 (*mod* means remainder of the integer division) and store it.
2. Calculate N *div* 2 (*div* means integer division).
3. If N *div* 2 is zero, then go to the step 4, otherwise assign N <- N *div* 2 and return to the step 1.
4. The stored remainders in the inverted order are the binary representation of the original N.

Example:

$75_{10} = (?)_2$

75 div 2 = 37 | 1

37 div 2 = 18 | 1

18 div 2 = 9 | 0

9 div 2 = 4 | 1

4 div 2 = 2 | 0

2 div 2 = 1 | 0

1 div 2 = 0 | 1

The direction remainders should be written in

$75_{10} = 1001011_2$

Alternative Method

1. Find the X = the greatest power of 2, which is less than N.
2. Write 1.
3. Assign N <- (N – X).
4. Repeat until N is 0:
   a. X <- X div 2
   b. Can X fit into N? if yes, write 1, and N <- N – X. If not, write 0.
5. The written 0 and 1 represent the original N in the binary system.

Example

75_{10} → the closest power of 2, which is less than 75, is 64:

X = 64, N = 11 | 1

X = 32, N = 11 | 0

X = 16, N = 11 | 0

X = 8, N = 3 | 1

X = 4, N = 3 | 0

X = 2, N = 1 | 1

X = 1, N = 0 | 1

75_{10} = 1001011_2

Binary to Decimal

1. Write the binary representation.
2. Below each digit write a respective power of two – start from the **rightmost digit** and write 1, then move towards left and write 2, then move towards left and write 4 until the leftmost digit of the binary number is reached.
3. Calculate the sum of powers, which have 1 above.

Example

$10101101_2 = (?)_{10}$

1	0	1	0	1	1	0	1
128	64	32	16	8	4	2	1
128	0	32	0	8	4	0	1

128 + 32 + 8 + 4 + 1 = 173

Binary to Hexadecimal

1. Group the binary digits into quadruples from right to left; add zeros before the first digit, if necessary.

2. Convert each quadruple to its hexadecimal representation.

$1011101111_2 = (?)_{16}$

0010 1110 1111

2 E F → $\mathbf{2EF_{16}}$

Hexadecimal to Binary

Just a reversed process – each digit is converted into its binary representation.

Decimal to Hexadecimal

Like the decimal to hexadecimal conversion, only the divisor is 16, not 2, and all remainder must be converted to hexadecimal version (if necessary).

Example

$428_{10} = (?)_{16}$

428 div 16 = 26 | C

26 div 16 = 1 | A

1 div 16 = 0 | 1

$\mathbf{428_{10} = 1AC_{16}}$

Hexadecimal to Decimal

Like binary to decimal except the powers (powers of 16) and before the final addition there must be calculated product of the digit and the respective power.

Basic Binary Arithmetic

Binary addition works in the same way like decimal one, except that only 0's and 1's can be used, instead of the whole spectrum of 0-9. This actually makes binary addition much simpler than decimal addition – there are possible these situations only:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 1 = 0 (and carry 1; 1+1 is 2 – in binary 10)
- 1 + 1 + 1 (carried) = 1 (and carry 1)

Example

```
  10011011
 +10011011
 100110110
```

Binary subtraction[1] can be calculated the very same way like the decimal one. There are 4 simple rules, which one must keep in the mind:

0 – 0 → 0
0 – 1 → 1, borrow 1
1 – 0 → 1
1 – 1 → 0

Subtracting a "1" digit from a "0" digit produces the digit "1", while 1 will have to be subtracted from the next column. This is known as borrowing. The principle is the same as for carrying. When the result of a subtraction is less than 0, the least possible value of a digit, the procedure is to "borrow" the deficit divided by the radix (that is, 10/10) from the left, subtracting it from the next positional value.

```
    *   * * *   (starred columns are borrowed from)
  1 1 0 1 1 1 0
−       1 0 1 1 1
  -------------
  1 0 1 0 1 1 1
```

Subtraction by 2's Complement

The subtraction in computer systems is done by addition of the minuend and the 2's complement of the subtrahend. 2's complement can be calculated in these steps:

1. Add 0s to the shorter number to make both numbers equally long (in computers it is implicit)
2. Invert digits (bits) of the binary number.
3. Add one.

When the addition is finished, the result must be trimmed to the length of the longer number.

Example

```
 10011011
    -1011
---------
 10011011
+11110101
---------
1|10010000|
```

(1011 is shorter ->add 0s -> 00001011 -> inversion -> 11110100 -> add 1 -> 11110101)

[1] http://en.wikipedia.org/wiki/Binary_division#Division

Binary multiplication[2] is actually much simpler than decimal multiplication. In the case of decimal multiplication, we need to remember 3 x 9 = 27, 7 x 8 = 56, and so on. In binary multiplication, we only need to remember the following:

```
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
```

Note that since binary operates in base 2, the multiplication rules we need to remember are those that involve 0 and 1 only. As an example of binary multiplication we have 101 times 11,

```
  101
  x11
```

First we multiply 101 by **1**, which produces 101. Then we shift and multiply 101 by **1**, which produces 101.

```
  101
  x11
  101
 101
```

The next step, as with decimal multiplication, is to add.

```
  101
  x11
  101
 101
 1111
```

Binary division is almost as easy, and involves knowledge of binary multiplication. Example: the division of 1011 into 11.

```
10110 / 11 = 111
-11
010
 101
 -11
  10
  100
  -11
  001
```

[2] http://www.binarymath.info/multiplication-division.php

`10110 / 11 = 111, remainder is 1`

To check our answer, we first multiply our divisor **11** by our quotient **111**. Then we add its' product to the remainder **1**, and compare it to our dividend of **10110**.

```
  111
x  11
  111
 111
10101
```
<-- product of 11 and 11

```
10101
+   1
10110
```
<-- sum of product and remainder

The sum is equal to our initial dividend, therefore our solution is correct.

Representation of Numbers

Binary Representation of Unsigned Integers

- The simplest case
  - Unsigned integers = non-negative values only – the lowest values is 0 (when all digits are 0),
  - the highest value depends on the number of bits, e.g. if there are 4 bits: $1111_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 15_{10}$,
  - In general *n* bits allow 2^n different combinations of 1s and 0s with values from 0 to 2^{n-1} (e.g. the *word* type has 16 bits, and its range is from 0 to $2^{16}-1$ = 65535) .
- Other useful facts
  - The minimal number of bits required to store a decimal number in the binary system is the whole part of the $\log_2(NUMBER) + 1$ - e.g. 86_{10} requires $[\log_2(86)] + 1$= 7 bits.
  - To determine, if a number is either odd or even, it is enough to check the least significant bit – if it is 1, it is an odd number, if 0 – even.

Binary Representation of Signed Integers

- There are several types of representation. The most common are
  - ***Sign magnitude*** – the first bit represents the sign (0 = +, 1 = -), the rest is the value. Human readable with slow calculations, redundant zero (with + and -); range of the values in N bits is $<-2^{N-1} + 1; 2^{N-1}-1>$.
  - ***One's complement*** – negative number is gained from its absolute value (non-negative representation) by inversion of all bits (e.g. -19 -> 19 is 0001 0011 in one byte -> -19 is 1110 1100 in the byte). More natural for computers, still there is a problem with 2 zeros (e.g. 0000 0000 and 1111 1111 in one byte); range of the values in N bits is $<-2^{N-1} + 1; 2^{N-1}-1>$.
  - ***Two's (or 2's) complement***
    - The default representation with fastest operations (automatically done by CPU).
    - The negative number is acquired from its non-negative value by calculation of the one's complement and addition of one (e.g. -19 -> 19 is 0001 0011 in one byte -> one's complement is 1110 1100 -> +1 -> 1110 1101). The backward conversion to its absolute value is done by the very same steps (e.g. 1110 1101 -> 0001 0010 -> +1 -> 0001 0011).
    - There is no redundant zero
    - Range of the values in N bits is $<-2^{N-1}; 2^{N-1}- 1>$.

Binary Representation of Real Numbers

Fixed Decimal Point

- Simple structure

N-bits for the integer part	M-bits for the fractional part

  E.g. 16 bits -> 12 bits for integer part, 4 bits for the decimal places -> the maximum is 4095,9375 for unsigned values (0,9375 = 1/2 + 1/4 + 1/8 + 1/16), the lowest non-zero value is 0,0625 = 1/16).
- Limited precision and range
- Used in cheap CPUs and microcontrollers

Floating Decimal Point

- Its structure is similar to the scientific notation of numbers

Sign	N-bits for the **exponent**	M-bits for the **mantissa**

- **Sign** – it indicates the value of the number (0 for non-negative, 1 for negative)
- **Mantissa** – significant digits of the number (e.g. *double* uses 52 bits)

- **Exponent** – a tricky part of the number – unsigned value; to get its actual meaning, a bias must be subtracted (e.g. in case of **DOUBLE** the exponent has 11 bits (values from 0 to 2047) and bias 1023; thus it can represent values <0-1023; 2047-1023>, i.e. <-1023; 1024>.
  It is complicated even more: exponent with zeros only and ones only indicate special values (zero, subnormal numbers, NaN – not a number, infinity).
- Reason: to simplify the hardware for comparing two exponents (to use simpler integer sorting rather than subtraction), we may want to avoid 2's complement representation for the exponent. This can be done by simply adding 1 (a bias) at the MSB of the exponent field and the resulting representation is called biased notation.
- The represented number has the value

$$Sign \times Mantissa \times 2^{Exponent-BIAS}$$

  The mantissa must be **NORMALIZED** – it must be a value from the interval <1; 2) - to efficiently use the bits available for the significand, it is shifted to the left until all leading 0's disappear (as they make no contribution to the precision)[3]. Because the value always starts by 1, this value is not included in the mantissa (it is implicitly added to the mantissa.

There are special values for the floating point representation:

SUBNORMAL NUMBERS

- any non-zero number which is smaller than the smallest acceptable number

[3] http://fourier.eng.hmc.edu/e85/lectures/arithmetic_html/node11.html

- subnormal numbers provide the guarantee that addition and subtraction of floating-point numbers never underflows; two nearby floating-point numbers always have a representable non-zero difference. Without gradual underflow, the subtraction A−B can underflow and produce zero even though the values are not equal.
- That would lead to DIVISION BY ZERO errors.
- represented by 0's in the exponent and non-zero value in the mantissa.

INFINITY

- used in case of overflow (the result is greater than the maximal acceptable number)
- both negative and positive infinity are possible
- represented by 1's only in the exponent and 0's in the significand

NAN (NOT A NUMBER)

- it is used to represent a value that does not represent a real number
- results of operations like 0 x ∞, log(-4), 0/0 …
- represented by 1's only in the exponent and a non-zero significand

Example

Representation of 651,322 – single precision (1 bit for sign, 8 bits for exponent, 23 bits for mantissa)

1. Normalised value: 1.27211328125 x 2^9
2. Sign in binary: 0
3. Exponent in binary: 0000 1001 → add bias (127) → 1000 1000
4. Mantissa: (implicit 1) + 01000101101010010011011

The final value: 01000100001000101101010010011011 – in hexadecimal: 44-22-D49B

For further reading, see http://en.wikipedia.org/wiki/Floating-point_number or http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/represent.html.

Exercise

1. Convert the unsigned numbers from the binary system to the decimal one
    a. 10100111101
    b. 1000101
    c. 0110111
    d. 11111111
2. Convert the numbers to the binary system
    a. $(67)_{10}$
    b. $(-12)_{10}$
    c. $(F2)_{16}$
    d. $(F0A1)_{16}$
3. Convert to binary
    a. -45 (8 bits)
    b. -2012 (16 bits)
    c. -1 (16 bits)
4. Multiply in binary
    a. 23 x 9
    b. 14 x 8
    c. 76 x (-25)
5. The picture represents so called binary clock
    a. Find out how the time is encoded in the clock (circles represent individual LEDs; the pink LEDs are on).
    b. Suggest a way how the clock might display also the number of the week day. How many lights/LEDs should be added or removed at least?