

Introduction to Digital Systems: Digital Encoding

Michał Kalisiak, Wiktor Porakowski
Warsaw University of Technology
Institute of Electronic Systems

We present the materials to learn/remind diverse encoding patterns. We discuss integers and real numbers, positive and negative. We encourage you to go through the examples with us on paper. You will find here some references to the C programming language as we want to keep it practical. We invite you to implement the proposed functions and observe the results.

If you notice any mistakes (even small) or something is unclear, please let me know:

Michal.Kalisiak@pw.edu.pl

List of topics

List of topics.....	1
Integers.....	2
BIN – HEX – OCT	3
Something about bit length.....	3
Data types in C.....	3
BONUS: C check.....	4
stdint.h	5
Unsigned.....	5
BIN, HEX, OCT to DEC	5
DEC to BIN	5
DEC to HEX, OCT	6
Addition, subtraction.....	6
Parameters	7
BONUS: C check.....	7
Bit print.....	7
Masks.....	8
Signed magnitude representation.....	9
BIN to DEC	10
DEC to BIN	10
Parameters	10
Two's complement representation.....	10
BIN to DEC	10
DEC to BIN	10
HEX, OCT.....	11
Addition, subtraction.....	11

Parameters	11
BONUS: C check.....	11
Real numbers.....	12
Fixed point.....	12
Unsigned.....	12
BIN to DEC	12
DEC to BIN	12
Rounding	13
Addition, subtraction.....	13
Parameters	13
Signed magnitude representation.....	13
BIN to DEC	14
DEC to BIN	14
Parameters	14
Signed (two's complement).....	14
BIN to DEC	14
DEC to BIN	14
Parameters	14
Floating point	15
Quarter precision float	15
BIN to DEC	15
DEC to BIN	16
Single precision float	18
BIN TO DEC	18
DEC to BIN	20
BONUS: C check.....	21
BONUS^2 endians.....	22
Homework.....	26

Integers

Let's start.

1011_B – it's the binary (BIN) number because it's written with subscript " B ". It has 4 bits (4 binary digits, so only zeros and ones).

10111011 – it's not the binary number. It's just 10 million 111 thousand 11. Decimal (DEC).

FA_H – that is the hexadecimal (HEX) number. HEX system has 16 different digits: 0,1, ... 9,A,B,C,D,E,F. $A_H = 10 = 1010_B$, ..., $F_H = 15 = 1111_B$.

72_O – that is octal (OCT) number. An octal system has 8 digits, from 0 to 7.

BIN – HEX – OCT

If we take a 9-bit number $1\ 1001\ 1110_B$, we can simply represent it in HEX. Just group it in 4 bits¹, starting from the least significant bit (LSB), i.e., from the right.

$$1_B = 1 = 1_H \quad 1001_B = 9 = 9_H \quad 1110_B = 14 = E_H$$

So $1\ 1001\ 1110_B = 19E_H$

To represent it in OCT we group it in three, analogously $110\ 011\ 110_B$

$$110_B = 6_O \quad 011_B = 3_O \quad 110_B = 6_O$$

So $110\ 011\ 110_B = 636_O$

To convert a number from HEX to OCT the simplest way is to go HEX-BIN-OCT, I think.

Something about bit length

If we design our own digital system, we can use as many (or as low) bits as we want to represent a number. E.g. we'll build a system that will add two 2-bit numbers and return a 3-bit number.

However, computers use 8, 16, 32, 64, ... bit numbers. Those can be signed or unsigned. Let's take the 8-bit number $1001\ 1110_B$. I have a question for you: is it signed or unsigned?

.

.

Well, we don't know. It depends on what we will say to the computer it is. So let's start with data types.

Data types in C

In C programming language, data ranges and bit size of common types like well-known "int" are not precisely defined, standard says: "implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign."² In the table below I show what the standard says and what I observe on my PC.

type	min size (bits)	minimal range	bits at my PC ³	the range at my PC ⁴
char	8	it depends		-128 ... 127
signed char	8	-127 ... 127		-128 ... 127
unsigned char	8	0 ... 255		
short	16	$-2^{15}+1 \dots 2^{15}-1$		$-2^{15} \dots 2^{15}-1$
short int		-32 767 ... 32 767		-32 768 ... 32 767
signed short				
signed short int				
unsigned short	16	$0 \dots 2^{16}-1$		

¹ why so? Because one digit in HEX is represented by four digits in BIN. $0_H, \dots, F_H \rightarrow 0000_B, \dots, 1111_B$

² ISO/IEC 9899:TC3, see the point 5.2.4.2.1. You can find it in our materials\Standards

³ if different

⁴ if different

unsigned short int		0 ... 65535		
int signed int	16	$-2^{15}+1 \dots 2^{15}-1$ -32 767 ... 32 767	32	$-2^{31} \dots 2^{31}-1$
unsigned int	16	$0 \dots 2^{16}-1$ 0 ... 65535	32	$0 \dots 2^{32}-1$
long long int	32	$-2^{31}+1 \dots 2^{31}-1$ -2147483647 ... 2147483647		$-2^{31} \dots 2^{31}-1$
unsigned long unsigned long int	32	$0 \dots 2^{32}-1$ 0 ... 4294967295		
long long long long int	64	$-2^{63}+1 \dots 2^{63}-1$		$-2^{63} \dots 2^{63}-1$
unsigned long long unsigned long long int	64	$0 \dots 2^{64}-1$		

BONUS: C check

You can check how your compiler behaves, here is an example of a function to do that

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void printLimits() {
    printf("--Integer limits--\n");
    printf("char: bits=%u, min=%i, max=%u\n", CHAR_BIT,
          CHAR_MIN, CHAR_MAX);
    printf("signed char: min=%i, max=%u\n", SCHAR_MIN,
          SCHAR_MAX);
    printf("unsigned char: max=%u\n\n", UCHAR_MAX);

    printf("short: bits=%u, min=%i, max=%u\n", sizeof(short)<<3,
          SHRT_MIN, SHRT_MAX);
    printf("unsigned short: max=%i\n\n", USHRT_MAX);

    printf("int: bits=%u, min=%i, max=%u\n", sizeof(int)<<3,
          INT_MIN, INT_MAX);
    printf("unsigned int: max=%u\n\n", UINT_MAX);

    printf("long int: bits=%u, min=%li, max=%lu\n", sizeof(long
          int)<<3, LONG_MIN, LONG_MAX);
    printf("unsigned long int: max=%lu\n\n", ULONG_MAX);

    printf("long long int: bits=%u, min=%lli, max=%llu\n",
          sizeof(long long int)<<3, LLONG_MIN, LLONG_MAX);
    printf("unsigned long long int: max=%llu\n\n", ULLONG_MAX);
}

int main(int argc, char** argv) {
    printLimits();
    return (EXIT_SUCCESS);
}
```

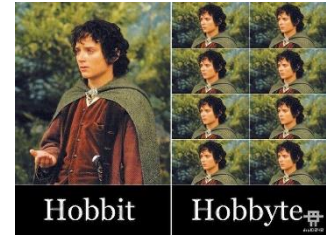
Btw, here we can see two nice operators, first is `sizeof()`, which returns the number of BYTES of variable or data type

```
sizeof(short) = 2
```

short has 2 bytes. Next, we do:

```
sizeof(short) << 3 so: 2 << 3 = 16
```

<< is the **left shift bitwise operator**. It shifts bits of 2 to the left by 3 positions. So $0000\ 0010_B \ll 3$ gives $0001\ 0000_B = 16$. It is the number of BITS of data type short! Very useful operator! He did multiplication $2 * 2^3$ for us!



stdint.h

If you project a digital system (e.g. on a microcontroller) often you would like to set precisely the bit size of a variable. So it is recommended to use (u)intB_t notation. For example, `uint8_t` is an 8-bit unsigned integer, `int32_t` is the 32 bit signed integer. If those names are unknown for the compiler add:

```
#include <stdint.h>.
```

Ok, so I think we are ready to go into bit representation!

Unsigned

To convert the unsigned number we just multiply its digits by powers of the notation base like in the decimal system

$$123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

BIN, HEX, OCT to DEC

$$\begin{aligned} 1001\ 1110_B &= 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \\ &= 128 + 16 + 8 + 4 + 2 = 158 \end{aligned}$$

From HEX we do similarly

$$1001\ 1110_B = 9E_H = 9 * 16^1 + E_H * 16^0 = 9 * 16^1 + 14 * 16^0 = 144 + 14 = 158$$

And with OCT

$$10\ 011\ 110_B = 236_O = 2 * 8^2 + 3 * 8^1 + 6 * 8_0 = 128 + 24 + 6 = 158$$

DEC to BIN

To convert decimal to binary

we divide by 2 and write the rest

$$\begin{aligned} 158/2 &= 79\ r\ 0 \\ 79/2 &= 39\ r\ 1 \\ 39/2 &= 19\ r\ 1 \\ 19/2 &= 9\ r\ 1 \\ 9/2 &= 4\ r\ 1 \\ 4/2 &= 2\ r\ 0 \\ 2/2 &= 1\ r\ 0 \\ 1/2 &= 0\ r\ 1 \end{aligned}$$



or simpler

$$\begin{array}{r|l} 158 & 0 \\ 79 & 1 \\ 39 & 1 \\ 19 & 1 \\ 9 & 1 \\ 4 & 0 \\ 2 & 0 \\ 1 & 1 \\ 0 & \end{array}$$



until we get 0. We obtain binary number from down to up (easy to remember, a starts normally from nonzero digit and at the end of dividing we always get 1!⁵) $158 = 1001\ 1110_B$

When we use an 8-bit unsigned number, we remember to fill empty MSB bits with zeros. E.g. after converting 2

$$\begin{array}{r|l} 2 & 0 \\ 1 & 1 \\ 0 & \end{array} \quad \uparrow$$

we write $0000\ 0010_B$.

DEC to HEX, OCT

To convert decimal to

HEX we do:
 $158/16 = 9\ r\ 14 = E_H$
 $9/16 = 0\ r\ 9$

and we get $158 = 9E_H$

OCT we do:
 $158/8 = 19\ r\ 6$
 $19/8 = 2\ r\ 3$
 $2/8 = 0\ r\ 2$

and we get $158 = 236_O$

Addition, subtraction

To add, the simplest is using the column method the same as in primary school, but easier, because we have only two digits! Let's add $158 + 86 = 244$

$$\begin{array}{r} 11\ 11 \\ 1001\ 1110 \\ +0101\ 0110 \\ \hline 1111\ 0100 \end{array}$$

Now let's add $158 + 99$

$$\begin{array}{r} 1\ 1111\ 1100 \\ 1001\ 1110 \\ +\ 0110\ 0011 \\ \hline 1\ 0000\ 0001 \end{array}$$

Let's check it in C using `uint8_t` data type

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h> // for uint8_t

int main(int argc, char** argv) {
    uint8_t a = 158;
    uint8_t b = 99;
    uint8_t c = a+b;
    printf("uint8_t %u + uint8_t %u = uint8_t %u", a, b, c);
    return (EXIT_SUCCESS);
}

>>> uint8_t 158 + uint8_t 99 = uint8_t 1
```

⁵ Except that later we have to fill empty most significant bits at the beginning with zeros, like in next example

1? Why? Because $158 + 99 = 257 = 1\ 0000\ 0001_B$ and it needs 9 bits to be stored! Adding of two N bit numbers can be $N + 1$ bit number. We should declare:

```
uint16_t c = (uint16_t)a + b;
```

Subtraction can be done analogically in the column method. And when we learn two's complement representation we'll see that we don't need subtraction at all!

Soon we'll learn how to build an adder using gates! Exciting, isn't it?

Parameters

The quantum (distance between two numbers) is 1. The minimum value is 0, the maximum for an 8-bit number is 255, for N bit number it is $2^N - 1$.

BONUS: C check

Bit print

In C there's no build-in function to print variable in binary representation, so we're going to make our own. We'll print `uint8_t` type, so 8-bit unsigned integer. Don't forget to `#include <stdint.h>`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h> // for uint8_t

void bitPrintf_uint8(uint8_t v){
    printf("%3u -> ", v);
    uint8_t mask = 1<<7;
    while(mask){
        printf( "%u", !(v & mask) );
        mask >>= 1;
    }
    printf("\n");
}

int main(int argc, char** argv) {
    uint8_t number = 156; // 10011100B
    bitPrintf_uint8(number);
    return (EXIT_SUCCESS);
}
```

So what happens in `bitprintf_uint8` function?

We make the 8-bit variable `mask` and set its most significant bit (MSB) to 1.

```
1<<7 = 128
```

Do you remember the left shift bitwise operator `<<` ?

$0000\ 0001_B \ll 7$ gives $1000\ 0000_B$. And now `while` our `mask` is not 0 we print something:

```
v & mask
```

`&` is another bitwise operator, called bitwise AND. It just makes AND of consecutive bits. So for $v = 1001\ 1100_B$, and $mask = 1000\ 0000_B$ we get

$$\begin{array}{r} 1001\ 1100 \\ \text{AND } 1000\ 0000 \\ \hline 1000\ 0000 \end{array}$$

Don't confuse **bitwise** AND operator & with **logical** AND operator &&.

```
!!(v&mask)
```

This is tricky, operator ! is **logical negation**. $0! = 1$, *something different from 0* $! = 0$, so in our case $1000\ 0000_B ! = 0$, and after another logical negation $0!$ we get 1, so the first digit of the binary representation of v . Btw, **bitwise negation** you use as $\sim v$.

$$\begin{array}{r} \sim 1001\ 1100 \\ \hline 0110\ 0011 \end{array}$$

```
mask >>= 1;
```

Now we use bitwise right shift **assignment**, which gives the same result as:

```
mask = (mask >>= 1)
```

that shifts bits of `mask` to the right, so $1000\ 0000_B \gg 1 = 0100\ 0000_B = 64$. So we divided by 2.

`while` loop goes on and on printing bits of our variable.

You can check all the bitwise operators here: https://en.wikipedia.org/wiki/Bitwise_operations_in_C

Masks

In low-level programming (e.g. microcontrollers) you often see something like that:

```
DDRC &= ~(1<<DDC1);
```

What is it? Maybe we'll start from the beginning. Just to make sure you understand bit shifting, we'll build a function like that:

```
#define BIT0 0
#define BIT1 1
#define BIT2 2
#define BIT3 3
#define BIT4 4
#define BIT5 5
#define BIT6 6
#define BIT7 7

void bitShift(){
    uint8_t REG;

    REG = 1<<BIT0; printf("REG = 1<<BIT0: "); bitPrintf_uint8(REG);
    REG = 1<<BIT1; printf("REG = 1<<BIT1: "); bitPrintf_uint8(REG);
    REG = 1<<BIT2; printf("REG = 1<<BIT2: "); bitPrintf_uint8(REG);
    REG = 1<<BIT3; printf("REG = 1<<BIT3: "); bitPrintf_uint8(REG);
    REG = 1<<BIT4; printf("REG = 1<<BIT4: "); bitPrintf_uint8(REG);
    REG = 1<<BIT5; printf("REG = 1<<BIT5: "); bitPrintf_uint8(REG);
    REG = 1<<BIT6; printf("REG = 1<<BIT6: "); bitPrintf_uint8(REG);
    REG = 1<<BIT7; printf("REG = 1<<BIT7: "); bitPrintf_uint8(REG);
}
```

and use it in our program.


```
>>>
REG = 1<<BIT0:      1 -> 00000001
REG = 1<<BIT1:      2 -> 00000010
REG = 1<<BIT2:      4 -> 00000100
REG = 1<<BIT3:      8 -> 00001000
REG = 1<<BIT4:     16 -> 00010000
REG = 1<<BIT5:     32 -> 00100000
REG = 1<<BIT6:     64 -> 01000000
REG = 1<<BIT7:    128 -> 10000000
```

Now imagine, that you want to configure a counter in your microcontroller, or analog to digital converter, or input register, or something. You have to set BIT1 and BIT5 in REG (**set** bit means to save 1_B, **clear** bit means to save 0_B, **toggle** means to change 0_B to 1_B and vice versa). Of course you can write REG = 0b00100010, but it's more elegant to write

```
uint8_t REG = (1<<BIT1) | (1<<BIT5);      bitPrintf_uint8(REG);
>>> 34 -> 00100010
```

we've used here bitwise OR operator here. After some operations of our microcontroller we have to **set** BIT7 and BIT0 in REG as well, but we don't want to change other bits. How to do it?

```
REG |= (1<<BIT0) | (1<<BIT7);              bitPrintf_uint8(REG);
>>> 163 -> 10100011
```

using bitwise OR **assignment** |=, which is the same as

```
REG = REG | (1<<BIT0) | (1<<BIT7);
```

Then, we need to **clear** BIT5. How to do that? No, 0<<BIT5 won't work. We prepare uint8_t mask with ones and BIT5=0 and do bitwise AND assignment &=

```
mask = ~(1<<BIT5);      printf("mask: ");   bitPrintf_uint8(mask);
REG &= mask;             printf("reg: ");     bitPrintf_uint8(REG);

>>> mask: 223 -> 11011111
>>> reg: 131 -> 10000011
```

and now we want to **toggle** BIT4 (twice, to check!)

```
                                bitPrintf_uint8(REG);
REG ^= (1<<BIT4); bitPrintf_uint8(REG);
REG ^= (1<<BIT4); bitPrintf_uint8(REG);

>>>
131 -> 10000011
147 -> 10010011
131 -> 10000011
```

^= is the bitwise XOR assignment. OK, let's get back to the bit representation of numbers. Now signed integers.

Signed magnitude representation

It is rather not used for integers. But it is used with floating-point numbers – you will find out later. The most significant bit (MSB) decides about the sign $MSB = 0 \rightarrow$ a number is positive, $MSB = 1 \rightarrow$ a number is negative. Rest of the bits we convert as unsigned.

BIN to DEC

For 8 bit binary number:

$$01010110_B = (-1)^0 * (1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0) \\ = 64 + 16 + 4 + 2 = 86$$

$$11010110_B = (-1)^1 * (1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0) = -86$$

DEC to BIN

In opposite direction, we normally convert like for unsigned and fill MSB with 1 for a negative or 0 for a positive number.

Parameters

The quantum is 1. The minimum value is $-(2^{N-1} - 1) = 111 \dots 1_B$, $\max(2^{N-1} - 1) = 011 \dots 1_B$, where N is bit length. Changing the sign of a number is very easy, however, adding is not. This representation includes two zeros $000 \dots 0_B = 0$ and $100 \dots 0_B = -0$.

Two's complement representation

BIN to DEC

Standard used commonly e.g. in computers. Looks like that:

$$01010110_B = -0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \\ = 64 + 16 + 4 + 2 = 86$$

$$11010110_B = -1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \\ = -128 + 64 + 16 + 4 + 2 = -128 + 86 = -42$$

We have to declare, that we use e.g. 8-bit numbers. It is very important.

DEC to BIN

To obtain -86 we calculate binary representation of 86 like before

86	0
43	1
21	1
10	0
5	1
2	0
1	1
0	

that is 1010110_B , but as I said, **it is very important to use 8 bits** so I should write:

$$86 = 01010110_B$$

Then we negate all bits

$$\overline{01010110}_B = 10101001_B = -87$$

and add 1_B

$$10101001_B + 1_B = 10101010_B = -86$$

That's simple!

Calculator: <https://www.exploringbinary.com/twos-complement-converter/>

HEX, OCT

If we want to represent signed BIN numbers in HEX or OCT, we do the same as for unsigned numbers. The only difference is we interpret them knowing the type. For a processor, they are only strings of zeros and ones (or levels of voltage).

Addition, subtraction

Adding stays the same as for unsigned numbers! That's a great message. And one more thing: to subtract $x - y = x + (-y) = x + \bar{y} + 1$.

Parameters

The quantum is 1. The minimum value is $-2^{N-1} = 100 \dots 0_B$, $\max(2^{N-1} - 1) = 011 \dots 1_B$, where N is bit length. It has one zero $00 \dots 0$.

BONUS: C check

We'll build very similar function to check signed `int8_t` type

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h> // for int8_t

void bitPrintf_int8(int8_t v){
    printf("%4i -> ", v);
    uint8_t mask=1<<((sizeof(v)<<3)-1);
    while(mask){
        printf("%u", !(v&mask));
        mask >>= 1;
    }
    printf("\n");
}

int main(int argc, char** argv) {
    int8_t number = -50;
    bitPrintf_int8(number);
}

>>> -50 -> 11001110
```

We can be afraid of what will happen when we shift bits. **Left** bitwise shifting is safe because we have free space on the right side

```
number <<= 1;                                bitPrintf_int8(number);

>>> -100 -> 10011100
```

But **right** shift? If we'll get 0 at most significant bit it would be a tragedy...

```
number >>= 2;                                bitPrintf_int8(number);

>>> -25 -> 11100111
```

Nothing like that happens! Compiler is very clever and puts ones at the beginning!⁶

⁶ eg. in Java there's a special the **unsigned** right shift operator `>>>` that puts zeros instead of ones regardless integer sign

Real numbers

Fixed point

Unsigned

To convert the unsigned fixed-point number we just multiply its digits by powers of the notation base like in the decimal system.

$$123.45 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2}$$

BIN to DEC

Before we do that we have to clarify how many bits the fractional part occupies. Do you remember? A processor knows only zeros and ones, it doesn't store dots! If we assume that dot is after $F = 3$ bits, we count:

$$\begin{aligned} 1\ 0011.110_B &= 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} \\ &= 16 + 2 + 1 + 0.5 + 0.25 = 19.75 \end{aligned}$$


I know you just said: "hmm no big deal, you just took the value from the previous example $1001\ 1110_B = 158$ and divided by $2^3 = 8$, nothing special!". You're right! It's nothing special. It is only representation.

DEC to BIN

To convert decimal 19.75 to binary with 3 bit fractional part, we can just multiply it by 2^3 and do the same as for unsigned (below, left⁷). However, sometimes it is easier (when the bit length is big, you'll see in floats) to do it in another way. We take the integer part and convert it normally. Then, we take the fractional part, multiply by 2 and when the outcome is bigger than 1, we generate 1 for the final result and continue multiplying with the rest like below, right. It is important to write bits in a proper direction. It is still easy to remember while dividing we take upside down (because – previous rule – number normally starts with a nonzero digit), and while multiplying we do the opposite so from up to down.

version 1.


158	0
79	1
39	1
19	1
9	0
4	0
2	0
1	1
0	



version 2.


integer part

19	1
9	1
4	0
2	0
1	1
0	



fractional part

$0.75 * 2 = 1.5 \rightarrow 1$
 $(1.5 - 1) * 2 = 1 \rightarrow 1$
and nothing remains, we have 3 bits for fractional, so, we add 0
 $\rightarrow 0$



⁷ [some time passed] I think version 1 is not always perfect. Sometimes (e.g. for floats) it is better to have more digits, especially when we don't know exactly how many significant bits do we need (see example in Quarter precision float->DEC to BIN), and round it. Version 1 does not allow that! Of course if we get perfect representation without rounding it does not matter.

Rounding

Now the task is to encode 4.7231 using fixed point, $N = 8$, $F = 4$. I'll use version 2.

$$4 = 0100_B$$

$$\begin{aligned} 0.7231 * 2 &= 1.4462 \rightarrow 1 \\ 0.4462 * 2 &= 0.8924 \rightarrow 0 \\ 0.8924 * 2 &= 1.7848 \rightarrow 1 \\ 0.7848 * 2 &= 1.5696 \rightarrow 1 \\ 0.5696 * 2 &= 1.1392 \rightarrow 1 \\ 0.1392 * 2 &= 0.2784 \rightarrow 0 \\ 0.2784 * 2 &= 0.5568 \rightarrow 0 \\ 0.5568 * 2 &= 1.1136 \rightarrow 1 \\ 0.1136 * 2 &= 0.2272 \rightarrow 0 \end{aligned}$$



wait, we don't have so many bits! We have to **round** it! $0.10111 \dots \cong 0.1100$ – like that

$$4.7231 \cong 0100.1100_B = 4 + 0.5 + 0.25 = 4.75$$

$$|error_1| = |4.75 - 4.7231| = 0.0269$$

The error should be no bigger than half of the quantum, i.e., $\frac{1}{2} * 2^{-4} = 2^{-5} = 0.03125$. It is ok. If this comparison fails, probably we did bad rounding.

E.g. if we just take the first 4 bits from multiplication 0100.1011_B , we get 4.6875 and $|error_1| = 0.0356 > 0.03125$ bigger than expected! Be careful!

What will happen if we use more bits? When we round to bit marked **green** we need to have one more bit calculated (verse $0.1136 * 2 = 0.2272 \rightarrow 0$) to round properly.

$$4.7231 \cong 0100.10111001_B = 4 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} = 4.72265625$$

$$error_2 = |4.72265625 - 4.7231| = 0.00044375$$

half of the quantum is $\frac{1}{2} * 2^{-8} = 0.00193125 \geq error_2$. OK.

Addition, subtraction

To add, the simplest is using the column method the same as in primary school, the only rule was to align dots! Here's the same rule

$$\begin{array}{r} 11 \\ 1001.1110 \\ +0101.01 \\ \hline 1111.0010 \end{array}$$

Parameters

The quantum (distance between two numbers) is 2^{-F} , where F is a bit length of a fractional part.

The minimum value is 0, the maximum for N bit number is $\frac{2^N - 1}{2^F}$.

Signed magnitude representation

As the fixed point is almost the same as the integer, this representation is also rather not used. We present it to preserve order.

BIN to DEC

$$1\ 0011.110_B = (-1)^1[0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3}]$$

$$= -[2 + 1 + 0.5 + 0.25] = -3.75$$

DEC to BIN

In opposite direction, we normally convert like for unsigned and fill MSB with 1 for a negative or 0 for a positive number.

Parameters

The quantum (distance between two numbers) is 2^{-F} , where F is a bit length of a fractional part.

The minimum value is $-\frac{2^{(N-1)}-1}{2^F} = 111 \dots 1.1 \dots 1_B$, where N is bit length, the maximum is $\frac{2^{(N-1)}-1}{2^F} = 011 \dots 1.1 \dots 1_B$. This representation includes two zeros $000 \dots 0.0 \dots 0_B = 0$ and $100 \dots 0.0 \dots 0_B = -0$.

Signed (two's complement)

BIN to DEC

$$1\ 0011.110_B = -1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3}$$

$$= -16 + 2 + 1 + 0.5 + 0.25 = -16 + 3.75 = -12.25$$

$$1\ 101.0110_B = -1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} + 0 * 2^{-4}$$

$$= -8 + 4 + 1 + 0.25 + 0.125 = -8 + 5.375 = -2.625 = -\frac{42}{2^4}$$

DEC to BIN

To convert -2.625 as an 8-bit binary number with the decimal point before 4 LSB ($F = 4$) we convert $+2.625$. Let's use version 2.

$$2 = 10_B$$

$$0.625 * 2 = 1.25 \rightarrow 1$$

$$0.25 * 2 = 0.5 \rightarrow 0$$

$$0.5 * 2 = 1 \rightarrow 1$$

and because $F = 4$ we add $\rightarrow 0$

$$+2.625 = 10.1010_B = 0010.1010_B$$

Then negation and "1". But this is one at the least significant bit, so in that case⁸ it would be 0.0001.

$$-2.625 = \overline{0010.1010}_B + 0.0001 = 1101.0101_B + 0.0001 = 1101.0110_B$$

Brilliant! (Don't forget about proper **rounding**, if needed!)

Parameters

The quantum (distance between two numbers) is 2^{-F} , where F is a bit length of a fractional part.

The minimum value for N bit number is $-\frac{2^{(N-1)}}{2^F} = 100 \dots 0.0 \dots 0_B$, the maximum for N bit number is $\frac{2^{(N-1)}-1}{2^F} = 011 \dots 1.1 \dots 1_B$. In this representation is one zero $00 \dots 0.0 \dots 0_B$.

⁸ once again: remember that processor doesn't have a dot, it uses only zeros and ones!

Floating point

I'm not sure if starting with the 32-bit IEEE 754 floating-point standard is a good idea. It would be easier to take a look at a smaller amount of bits but keeping all the properties.

Quarter precision float

The idea is like that: we have 8 bits, MSB is the sign (acting like in signed magnitude representation), then we have 3 bits of exponent and 4 bits of significand

$$float = s e_2 e_1 e_0 m_1 m_2 m_3 m_4$$

where

$$e = e_2 * 2^2 + e_1 * 2^1 + e_0 * 2^0$$

$$m = m_1 * 2^{-1} + m_2 * 2^{-2} + m_3 * 2^{-3} + m_4 * 2^{-4}, m \in [0; 1) \text{ (the fixed point encoding here!).}$$

BIN to DEC

Decimal value val represented by this $float$ string depends on values stored in e

1. $e \neq 0$ and $e \neq 111_B$, then $val_1 = (-1)^s * 2^{e-3} * (1 + m)$. It is the most common case.
2. $e = 0$, then $val_2 = (-1)^s * 2^{-2} * m$
 - a. in particular, for $m = 0$, $val_{2a} = (-1)^s * 0$, signed zero.

and special cases:

3. $e = 111_B$ and $m = 0$, then $val_3 = (-1)^s * \infty$
4. $e = 111_B$ and $m \neq 0$, then $val_4 = NaN$ (not a number)

I think it would be good to show a table with all the possibilities to see what happens here. We can observe only half of the possible values, for $s = 0$, because for $s = 1$ it would be the opposite value.

Notice, that the first 16 values we count for case no. 2. For others (up to $e = 111_B$) we use case no. 1. and later we have special cases. Values are monotonically growing, with growing quantum which depends on exponent value. When you understand that pattern, IEEE 754 standard will be super-easy!

<i>float</i>	<i>s</i>	<i>e</i>	<i>m</i>	<i>val</i>	quantum
0	0	000 _B	0000 _B	$val_{2a} = (-1)^0 * 0 = 0$	$2^{-6} = 0.015625$
1	0	000 _B	0001 _B	$val_2 = (-1)^0 * 2^{-2} * 2^{-4} = 2^{-6} = 0.015625$	
...					
15	0	000 _B	1111 _B	$val_2 = 2^{-2} * [2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}] = 2^{-2} * 0.9375 = 0.234375$	$2^{-6} = 0.015625$
16	0	001 _B	0000 _B	$val_1 = 2^{1-3} * (1 + 0) = 2^{-2} * 1 = 0.25$	
17	0	001 _B	0001 _B	$val_1 = 2^{-2} * (1 + 2^{-4}) = 2^{-2} + 2^{-6} = 0.265625$	
...					$2^{-6} = 0.015625$
31	0	001 _B	1111 _B	$val_1 = 2^{-2} * (1 + [2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}]) = 2^{-2} * 1.9375 = 0.484375$	
32	0	010 _B	0000 _B	$val_1 = 2^{2-3} * (1 + 0) = 2^{-1} * 1 = 0.5$	
33	0	010 _B	0001 _B	$val_1 = 2^{-1} * (1 + 2^{-4}) = 2^{-1} + 2^{-5} = 0.53125$	$2^{-5} = 0.03125$
...					
47	0	010 _B	1111 _B	$val_1 = 2^{-1} * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^{-1} * 1.9375 = 0.96875$	
48	0	011 _B	0000 _B	$val_1 = 2^{3-3} * (1 + 0) = 2^0 * 1 = 1$	$2^{-4} = 0.0625$
49	0	011 _B	0001 _B	$val_1 = 2^0 * (1 + 2^{-4}) = 2^0 + 2^{-4} = 1.0625$	

...					
63	0	011 _B	1111 _B	$val_1 = 2^0 * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^0 * 1.9375 = 1.9375$	
64	0	100 _B	0000 _B	$val_1 = 2^{4-3} * (1 + 0) = 2^1 * 1 = 2$	$2^{-3} = 0.125$
65	0	100 _B	0001 _B	$val_1 = 2^1 * (1 + 2^{-4}) = 2^1 + 2^{-3} = 1.125$	
...					
79	0	100 _B	1111 _B	$val_1 = 2^1 * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^1 * 1.9375 = 3.875$	
80	0	101 _B	0000 _B	$val_1 = 2^{5-3} * (1 + 0) = 2^2 * 1 = 4$	$2^{-2} = 0.25$
81	0	101 _B	0001 _B	$val_1 = 2^2 * (1 + 2^{-4}) = 2^2 + 2^{-2} = 4.25$	
...					
95	0	101 _B	1111 _B	$val_1 = 2^2 * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^2 * 1.9375 = 7.75$	
96	0	110 _B	0000 _B	$val_1 = 2^{6-3} * (1 + 0) = 2^3 * 1 = 8$	$2^{-1} = 0.5$
97	0	110 _B	0001 _B	$val_1 = 2^3 * (1 + 2^{-4}) = 2^3 + 2^{-1} = 8.5$	
...					
111	0	110 _B	1111 _B	$val_1 = 2^3 * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^3 * 1.9375 = 15.5$	
112	0	111 _B	0000 _B	$val_3 = (-1)^0 * \infty = +\infty$	
113	0	111 _B	0001 _B	$val_4 = NaN$	
...					
127	0	111 _B	1111 _B		

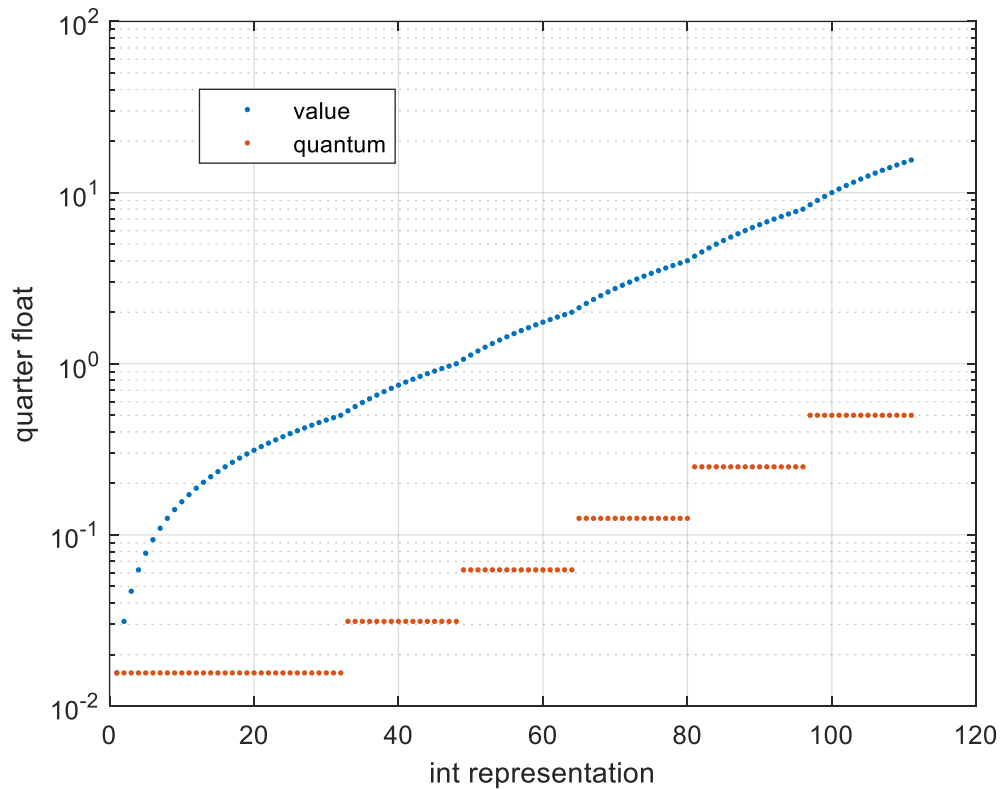


Fig. 1. Value and quantum of quarter precision float for positive values (the table in the figure).

DEC to BIN

Our number has to be in the range of the data type, so $[-15.5; 15.5]$. E.g. -2.7391 .

1. Immediately we set the bit of sign, $-2.7391 < 0 \rightarrow s = 1$.

2. We write DEC number in fixed point encoding (absolute value)

$$2 = 10_B$$

$$0.7391 * 2 = 1.4782 \rightarrow 1$$

$$0.4782 * 2 = 0.9564 \rightarrow 0$$

$$0.9564 * 2 = 1.9128 \rightarrow 1$$

$$0.9128 * 2 = 1.8256 \rightarrow 1$$

$$0.8256 * 2 = 1.6512 \rightarrow 1$$

$$2.7391 \cong 10.10111$$

3. Now a very important step – normalization. This value should be from the range $[1; 2)$, as $10.10111_B = 1 + m$, so we write

$$10.10111_B = 1.010111_B * 2^1$$

Here we multiply the binary number by $2^{\text{something}}$, *something* must be in the set $\{-2, -1, 0, 1, 2, 3\}$. The normalization of m , so that $m \in [0; 1)$ and thus $1 + m \in [1; 2)$ is just the rule of IEEE 754 floating-point.

And a simple parallel to the decimal system: $123.4 = 1.234 * 10^2$; $0.001234 = 1.234 * 10^{-3}$

4. and we can extract exponent e here, as $e - 3 = 1$, thus $e = 4 = 100_B$ (quick look at the table to check if that's reasonable. Yes, it is!).
5. we pick the proper equation. Considering that $e \neq 0$ and $e \neq 111_B$, we choose

$$val_1 = (-1)^s * 2^{e-3} * (1 + m)$$

6. Now we subtract 1 from $1 + m = 1.010111_B$ and we've got

$$m = 0.010111_B$$

7. Almost at home! m occupies only 4 bits, so we should round it *properly*.

$$m = 0.\overbrace{01110}^{4 \text{ bits}}_B$$

We'll omit "0." in the next step – it is always there, it doesn't bring any information.

8. Perfect! Now we compose all the float string and we remember to omit "0." when putting m :

$$float = \overbrace{1}^s \overbrace{100}^e \overbrace{01110}^m_B$$

Let's check the encoding. We identify s, e, m . And calculate the value with the proper equation

$$s = 1$$

$$e = 100_B = 4$$

$$m = 0.01110_B = 2^{-2} + 2^{-3} = 0.375$$

$$val_1 = (-1)^1 * 2^{4-3} * (1 + 0.375) = -2 * 1.375 = -2.75$$

$$|error| = |-2.75 - (-2.7391)| = 0.0109$$

half of the quantum for $e = 4$ is $\frac{1}{2} * 2^{-3} = 0.0625$, so it is good.

Again, if we round improperly, we get

$$m = 0.0101_B ; float = 1\ 100\ 0101_B$$

$$val_1 = (-1)^1 * 2^{4-3} * (1 + [2^{-2} + 2^{-4}]) = -2 * 1.3125 = -2.627$$

$$|error| = |-2.625 - (-2.7391)| = 0.1141 > \frac{1}{2} quant$$

which proves that the solution is wrong. Of course, we don't have to check quantum and error, if we round properly.

Single precision float

Ok, now the icing on the cake – IEEE 754 float⁹. I promised that when you understand quarter-float it would be easy. Let's find out!

For single-precision float: we have 32 bits, MSB is the sign (acting like in signed magnitude representation), then we have 8 bits of exponent, and 23 bits of significand

$$float = s\ e_7 e_6 \dots e_0\ m_1 m_2 \dots m_{23}$$

where

$$e = e_7 * 2^7 + e_6 * 2^6 + \dots + e_0 * 2^0$$

$$m = m_1 * 2^{-1} + m_2 * 2^{-2} + \dots + m_{23} * 2^{-23},\ m \in [0; 1) \text{ (fixed-point here!)}.$$

BIN TO DEC

Value *val* represented by this *float* string depends on values stored in *e*

1. $e \neq 0$ and $e \neq 255 = 11 \dots 1_B$, then $val_1 = (-1)^s * 2^{e-127} * (1 + m)$. It is the most common case.
2. $e = 0$, then $val_2 = (-1)^s * 2^{-126} * m$
only for very small numbers of $|val_2| \leq 2^{-126} - 2^{-149} \cong 1.175e - 38$
 - a. in particular, for $m = 0$, $val_{2a} = (-1)^s * 0$, signed zero.

and special cases:

3. $e = 255 = 11 \dots 1_B$ and $m = 0$, then $val_3 = (-1)^s * \infty$
4. $e = 255 = 11 \dots 1_B$ and $m \neq 0$, then $val_4 = NaN$ (not a number)

I prepared a fragment of a **new table at the end of the document**, please have a look.

And here is a very nice converter <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Ok, so in the table, we've prepared a lot of conversions from BIN to DEC and checked them with the converter. Figures below present float values and quantum over the whole range and zoom on the first values.

⁹ IEEE Std 754, IEEE Standard for Floating-Point Arithmetic. Available in materials\Standards (only for IDS students!!)

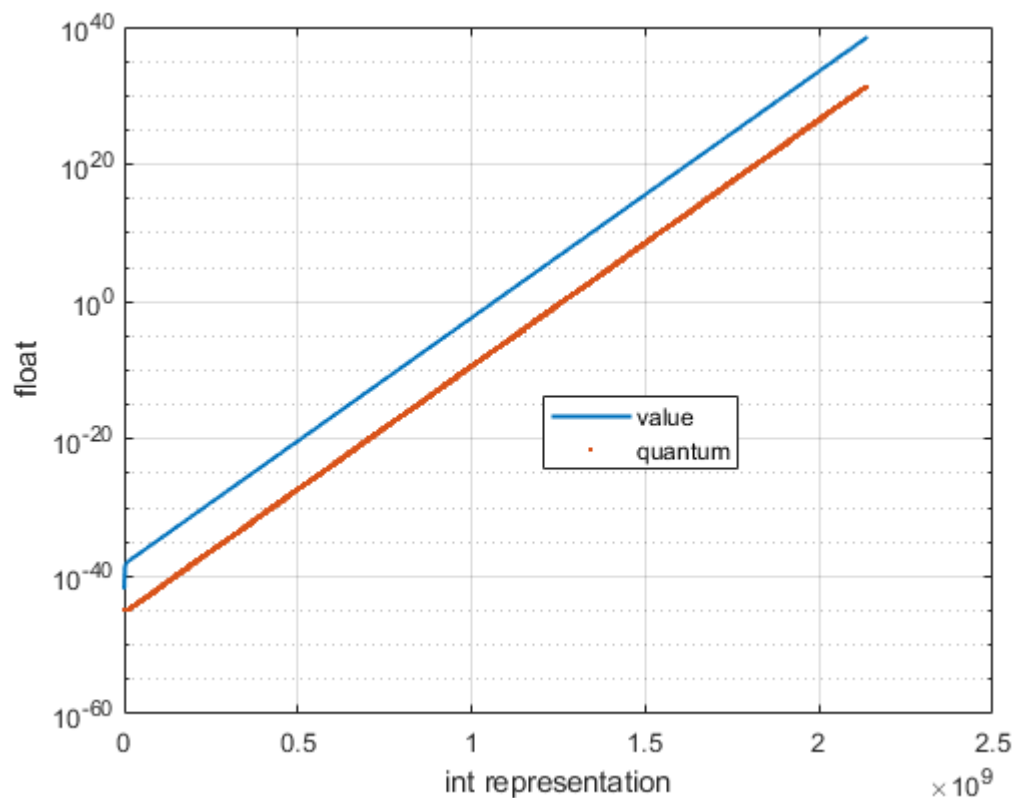


Fig. 2. Value and quantum of IEEE 754 float over the whole range of positive values.

Looking in fig. 2 – what will happen if we do an operation like that: $(b + s) - b$, where b – a big number, s – a small number?

.

.

We can completely lose information about s and the result of such operation would be 0. You can check it in the program.

```
float s = 2e-10;
float b = 2e+10;
printf("\n\n");
printf("small = %.10e\n", s); //scientific notation
printf("big   = %.10e\n", b);
printf("(big + small) - big = %.10e\n", (b+s)-b);

>>> small = 2.00000000267e-010
>>> big   = 2.00000000000e+010
>>> (big + small) - big = 0.00000000000e+000
```

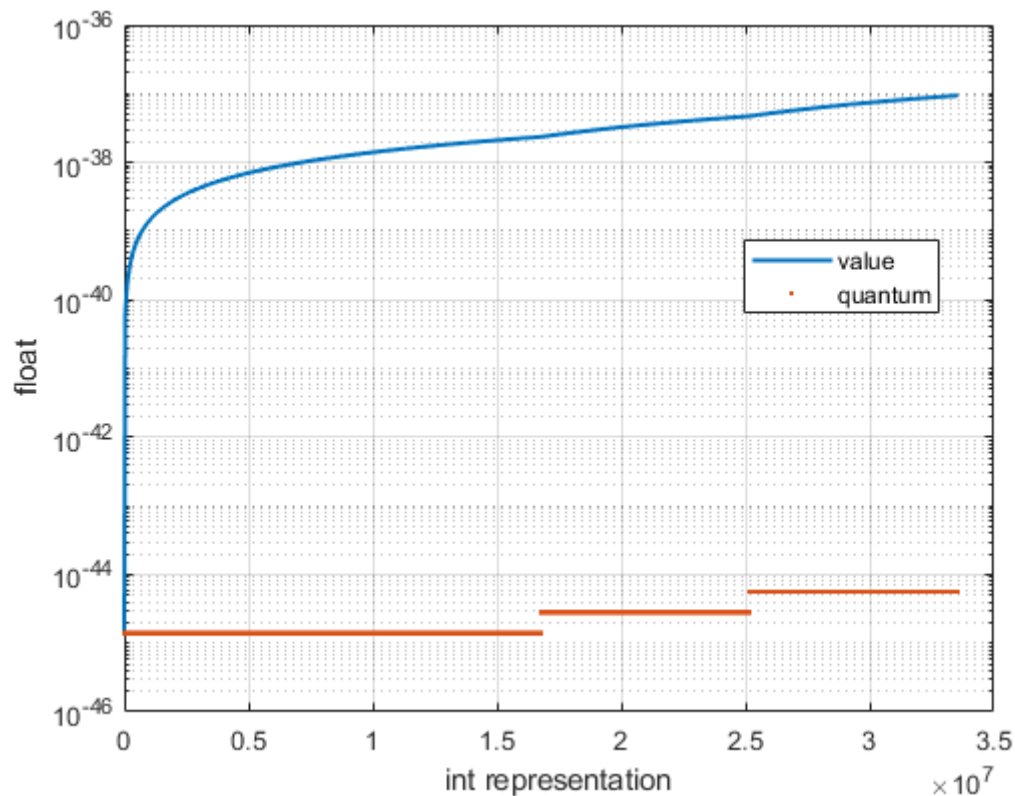


Fig. 3. Value and quantum of IEEE 754 float for first 4 value of exponent
 $float = [0, 2^{25} - 1]$ (int representation)

DEC to BIN

We do the same steps as for quarter-float. Number to convert: -18.25

1. set bit of sign

$$s = 1$$

2. write DEC number in fixed point encoding

integer part

18	0
9	1
4	0
2	0
1	1
0	

fractional part

$0.25 * 2 = 0.5 \rightarrow 0$
$0.5 * 2 = 1 \rightarrow 1$

$$18.25 = 10010.01_B$$

3. normalize the number to the range of $[1; 2)$

$$18.25 = 10010.01_B = 1.001001_B * 2^4$$

4. extract the exponent

$$e - 127 = 4 \rightarrow e = 131$$

131	1
65	1
32	0
16	0
8	0
4	0
2	0
1	1
0	

$$e = 1000\ 0011_B$$

5. we pick the proper equation. Exponent is not equal to 0 and not equal to 256, so we choose

$$val_1 = (-1)^S * 2^{e-127} * (1 + m)$$

6. get rid of 1 at MSB – it is needed for val_1 case to get m from $1 + m = 1.001001_B$

$$m = 0.001001_B$$

7. set/round significant m to the proper bit length

$$m = 0.\overbrace{001\ 0010\ 0000\ 0000\ 0000\ 0000}^{23\ bits}_B$$

8. write float representation (remembering to omit “0.” from m)

$$float = s\ e_7 e_6 \dots e_0\ m_1 m_2 \dots m_{23} = \overbrace{1}^s \overbrace{1000\ 0011}^e \overbrace{001\ 0010\ 0000\ 0000\ 0000\ 0000}^m_B$$

voilà!

Let’s check the encoding. We identify s , e , m . And calculate the value with the proper equation

$$s = 1$$

$$e = 1000\ 0011_B = 2^7 + 2^1 + 2^0 = 128 + 2 + 1 = 131$$

$$m = 0.001\ 0010\ 0000\ 0000\ 0000\ 0000_B = 2^{-3} + 2^{-6} = 0.140625$$

$$val_1 = (-1)^1 * 2^{131-127} * (1 + 0.140625) = -2^4 * 1.140625 = 18.25$$

It works.

The conversion would not be so pleasant if we’ll take e.g number 0.7. Why?

If you want, you can check the link with good examples

<http://sandbox.mc.edu/~bennet/cs110/flt/dtof.html>

and (also facultative) video how to add floats <https://www.youtube.com/watch?v=liHK18n0pm4>

BONUS: C check

Floats have not bit shifting. To check float bit representation we can do some tricks with data typecasting, or do something better – function for all the types of numbers¹⁰!

```
void bitPrintf(void *ptr, size_t N){
    uint8_t *array = ptr;
```

¹⁰ I’ve borrowed the code from here: <https://stackoverflow.com/questions/1697425/how-to-print-out-each-bit-of-a-floating-point-number>

```

uint8_t mask;
if (ptr == NULL)
    return;

while (N > 0) {
    --N;
    for(mask = 1<<7; mask; mask >>= 1)
        printf("%x", !(array[N] & mask));
    }
    printf("\n");
}

int main(int argc, char** argv) {
    float number = 18.25;
    bitPrintf( &number, sizeof(number) );
}

>>> 01000001100100100000000000000000

```

We pass pointer without a type to the function `void *ptr`, and BYTE size of a variable `size_t N`. So now we know inside the function where the value is stored in the memory, and how many bytes it occupies. We treat the number as an array of 8-bit unsigned numbers. So float 18.25 would be stored like:

<u>array[3]</u>	<u>array[2]</u>	<u>array[1]</u>	<u>array[0]</u>
0100 0001	1001 0010	0000 0000	0000 0000

In `while` loop we take consecutive bytes starting from `array[3]` and in `for` loop we do bitwise shifting, masking with bitwise AND, logical negation `!`, – everything is clear after chapter “Masks”, I hope!

BONUS^2 endians

When I was writing that I had something: “I have to check if I’m not lying here” – because pointers were always a little bit mysterious for me. So I checked the process of storing floats in RAM in my computer, if you want, you can follow me in this little very-bonus chapter.

I declare 3 types of pointers

```

float    *ptr_float = NULL;
void     *ptr_void  = NULL;
uint8_t  *ptr_uint8 = NULL;

```

I check the size which they (the pointers!) take in the memory

```

printf("sizeof(ptr_float) = %u   \n", sizeof(ptr_float));
printf("sizeof(ptr_void)   = %u   \n", sizeof(ptr_void));
printf("sizeof(ptr_uint8)  = %u \n\n", sizeof(ptr_uint8));

>>> sizeof(ptr_float) = 4
>>> sizeof(ptr_void)  = 4
>>> sizeof(ptr_uint8) = 4

```

The size of a pointer depends on whether the program is compiled as 32-bit (x86) or 64-bit (x64). Here, as the size of a pointer is 4 bytes, it is 32-bit. As proof, I attach screenshots from MS Visual Studio, where you can change the bit version of the application¹¹.

¹¹ courtesy of one of the students

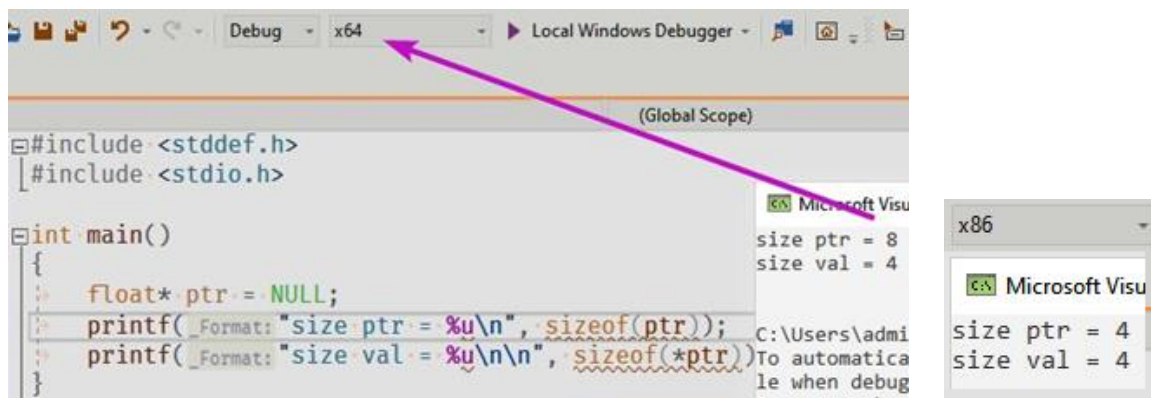


Fig. 4. The size of a pointer in 64-bit (x64) application – left, and in 32-bit (x86) application – right.

Now the size of data where the pointer points

```
printf("sizeof(*ptr_float) = %u \n", sizeof(*ptr_float));
printf("sizeof(*ptr_void) = %u \n", sizeof(*ptr_void));
printf("sizeof(*ptr_uint8) = %u \n\n", sizeof(*ptr_uint8));

sizeof(*ptr_float) = 4
sizeof(*ptr_void) = 1
sizeof(*ptr_uint8) = 1
```

No surprises here, 32-bit float has 4 bytes, void assumes 1 byte, and of course 1 byte for uint8_t. Then I check addresses where pointers point, and what happens when I add +1 to the address.

```
ptr_float = &number; // set pointer on the float number 18.25
ptr_void = ptr_float; // no warning here!
ptr_uint8 = ptr_void;

printf("ptr_float = %u, ptr_float + 1 = %u \n", ptr_float, ptr_float + 1);
printf("ptr_void = %u, ptr_void + 1 = %u \n", ptr_void, ptr_void + 1);
printf("ptr_uint8 = %u, ptr_uint8 + 1 = %u \n\n", ptr_uint8, ptr_uint8 + 1);

>>>ptr_float = 6422288, ptr_float + 1 = 6422292
>>>ptr_void = 6422288, ptr_void + 1 = 6422289
>>>ptr_uint8 = 6422288, ptr_uint8 + 1 = 6422289
```

That is what we'd expect. Now let's test array-shape. This: `ptr_uint8[3]` is not a pointer, this is value where pointer points, to get address, I add &

```
printf("&ptr_uint8[3] = %u \n", &ptr_uint8[3]);
printf("&ptr_uint8[2] = %u \n", &ptr_uint8[2]);
printf("&ptr_uint8[1] = %u \n", &ptr_uint8[1]);
printf("&ptr_uint8[0] = %u \n\n", &ptr_uint8[0]);

>>> &ptr_uint8[3] = 6422291
>>> &ptr_uint8[2] = 6422290
>>> &ptr_uint8[1] = 6422289
>>> &ptr_uint8[0] = 6422288
```

Looks great! So only one thing left, to check what's hidden under those addresses. We can use our trustworthy function `bitPrintf_uint8()`

```
printf("ptr_uint8[3] = "), bitPrintf_uint8(ptr_uint8[3]);
printf("ptr_uint8[2] = "), bitPrintf_uint8(ptr_uint8[2]);
```

```

printf("ptr_uint8[1] = "), bitPrintf_uint8(ptr_uint8[1]);
printf("ptr_uint8[0] = "), bitPrintf_uint8(ptr_uint8[0]);

ptr_uint8[3] = 65 -> 01000001
ptr_uint8[2] = 146 -> 10010010
ptr_uint8[1] = 0 -> 00000000
ptr_uint8[0] = 0 -> 00000000

```

So as you can see, under higher address, there are higher bits of data stored. This type of data storing is called **little-endian**. The opposite is called **big-endian**. I'll better show it in the table...

18.25 = 0100 0001 1001 0010 0000 0000 0000 0000

address:	&ptr_uint8[3] = 6422291	&ptr_uint8[2] = 6422290	&ptr_uint8[1] = 6422289	&ptr_uint8[0] = 6422288
little endian value	01000001	10010010	00000000	00000000
big endian value	00000000	00000000	10010010	01000001

Wikipedia¹² says: "The Intel x86 and AMD64 / x86-64 series of processors use the little-endian format." I'm wondering if your computers use little-endian as well.

¹² <https://en.wikipedia.org/wiki/Endianness>

<i>float</i>	<i>s</i>	<i>e</i>	<i>m</i>	<i>val</i>	quantum
00000000 _H	0	0	00 ... 00 _B	$val_{2a} = (-1)^0 * 0 = 0$	$2^{-149} \cong 1.40 * 10^{-45}$ ultra-small quantum
00000001 _H	0	0	00 ... 01 _B	$val_2 = (-1)^0 * 2^{-126} * 2^{-23} = 2^{-149} \cong 1.40 * 10^{-45}$	
...					
007FFFFFF _H	0	0	11 ... 11 _B	$val_2 = 2^{-126} * [2^{-1} + 2^{-2} + \dots + 2^{-23}] = 2^{-126} * [1 - 2^{-23}] = 2^{-126} - 2^{-149}$	
00800000 _H	0	1	00 ... 00 _B	$val_1 = 2^{1-127} * (1 + 0) = 2^{-126} \cong 1.18 * 10^{-38}$	$2^{-149} \cong 1.40 * 10^{-45}$
00800001 _H	0	1	00 ... 01 _B	$val_1 = 2^{1-127} * (1 + 2^{-23}) = 2^{-126} + 2^{-149}$	
...					
00FFFFFF _H	0	1	11 ... 11 _B	$val_1 = 2^{1-127} * (1 + [2^{-1} + 2^{-2} + \dots + 2^{-23}]) = 2^{-126} * (2 - 2^{-23}) = 2^{-125} - 2^{-149}$	
01000000 _H	0	2	00 ... 00 _B	$val_1 = 2^{2-127} * (1 + 0) = 2^{-125} \cong 2.35 * 10^{-38}$	$2^{-148} \cong 2.80 * 10^{-45}$
01000001 _H	0	2	00 ... 01 _B	$val_1 = 2^{2-127} * (1 + 2^{-23}) = 2^{-125} + 2^{-148}$	
...					
017FFFFFF _H	0	2	11 ... 11 _B	$val_1 = 2^{2-127} * (1 + [2^{-1} + 2^{-2} + \dots + 2^{-23}]) = 2^{-125} * (2 - 2^{-23}) = 2^{-124} - 2^{-148}$	
...					$2^{-23} \cong 1.19 * 10^{-7}$ not so accurate
3F800000 _H	0	127	00 ... 00 _B	$val_1 = 2^{127-127} * (1 + 0) = 2^0 = 1$	
3F800001 _H	0	127	00 ... 01 _B	$val_1 = 2^{127-127} * (1 + 2^{-23}) = 2^0 + 2^{-23}$	
...					
3FFFFFF _H	0	127	11 ... 11 _B	$val_1 = 2^0 * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^0 * (2 - 2^{-23}) = 2 - 2^{-23}$	$2^{103} \cong 1.01 * 10^{31}$ enormous quantum!!
...					
7E800000 _H	0	253	00 ... 00 _B	$val_1 = 2^{253-127} * (1 + 0) = 2^{126} \cong 8.51 * 10^{37}$	
7E800001 _H	0	253	00 ... 01 _B	$val_1 = 2^{253-127} * (1 + 2^{-23}) = 2^{126} + 2^{103}$	
...					$2^{104} \cong 2.03 * 10^{31}$
7EFFFFFF _H	0	253	11 ... 11 _B	$val_1 = 2^{253-127} * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^{126} * (2 - 2^{-23}) = 2^{127} - 2^{103}$	
7F000000 _H	0	254	00 ... 00 _B	$val_1 = 2^{254-127} * (1 + 0) = 2^{127} \cong 1.70 * 10^{38}$	
7F000001 _H	0	254	00 ... 01 _B	$val_1 = 2^{254-127} * (1 + 2^{-23}) = 2^{127} + 2^{104}$	
...					
7F7FFFFFF _H	0	254	11 ... 11 _B	$val_1 = 2^{254-127} * (1 + [2^{-1} + \dots + 2^{-4}]) = 2^{127} * (2 - 2^{-23}) = 2^{128} - 2^{104} \cong 3.40 * 10^{38}$	
7F800000 _H	0	255	00 ... 00 _B	$val_3 = (-1)^0 * \infty = +\infty$	
7F800001 _H	0	255	00 ... 01 _B	$val_4 = NaN$	
...					
7FFFFFF _H	0	255	11 ... 11 _B		

Homework

Please write ALL the calculations step by step. Tasks without explanation will not be rated. Prepare it on the computer or take a photo and upload your homework in Teams Assignment.

Name:

Index:

Write 3 last digits of your index

$$i_3 = \dots\dots\dots$$

count remainder of i_3 divided by 64 (modulo)

$$i_m = i_3 \% 64 = \dots\dots\dots$$

add 50 to i_m

$$x = i_m + 50 = \dots\dots\dots$$

Count also

$$y = 0.03125 * x = \dots\dots\dots$$

and

$$z = -0.1177 * x = \dots\dots\dots$$

E.g. if your index is 123456, then $i_3 = 456$, $i_m = 456 \% 64 = 8$, $x = 8 + 50 = 58$,
 $-x = -58$, $y = 0.03125 * 58 = 1.8125$, $z = -0.1177 * 58 = -6.8266$

1. Write x in binary (x_B), octal (x_O) and in hexadecimal (x_H) numerical system. For x_B use 8 bits (lead with zeros if necessary) 0.5 p
2. Convert $-x$ to signed integer in 8 bits two's complement representation ($-x_B$) 0.5 p
3. Add (using column method) $d_B = -x_B + x_B$ (use 8 bits for d_B) and interpret d_B in decimal system 0.25 p
4. Assume that **5** least significant bits (LSB) of $-x_B$ are the fractional part (**bbb.bbbbbb** is 8 bit **two's complement signed** fixed-point number) and convert it to real number 0.75 p
5. Convert y to 8 bit **two's complement signed** fixed-point binary number. The fractional part occupies **5 LSB (bbb.bbbbbb)**. 0.75 p
6. Convert z to 8 bit **two's complement signed** fixed-point binary number. The fractional part occupies **3 LSB (bbbbb.bbb)**. Remember about wise rounding! 0.75 p
7. Convert y to IEEE 754 float. 2 p
8. Convert your result from task 7. back to decimal representation. 0.5 p

Good luck!