Informatik - Exercise Session
Numerical representation

Introduction and Repetition

1. Perform the following steps:
   1.1 Convert the integer numbers $a = 4$ and $b = 7$ into their binary representation.
   1.2 Add the binary representations.
   1.3 Convert the result into decimal.
2. Evaluate the following expressions:
   2.1 `5 < 4 < 1`
   2.2 `true > false`

Introduction and Repetition - Solutions

1.
   1.1 $4_{10} = 100_2$ and $7_{10} = 111_2$
   1.2 $100_2 + 111_2 = 1011_2$
   1.3 $1011_2 = 11_{10}$

Introduction and Repetition - Solutions

1.
   1.1 $4_{10} = 100_2$ and $7_{10} = 111_2$
   1.2 $100_2 + 111_2 = 1011_2$
   1.3 $1011_2 = 11_{10}$
2.
   2.1

```
5 < 4 < 1
(5 < 4) < 1
false < 1
0 < 1
true
```

Introduction and Repetition - Solutions

1.
   1.1 $4_{10} = 100_2$ and $7_{10} = 111_2$
   1.2 $100_2 + 111_2 = 1011_2$
   1.3 $1011_2 = 11_{10}$
2.
   2.1

```
5 < 4 < 1
(5 < 4) < 1
false < 1
0 < 1
true
```

   2.2

```
true > false
true > 0
1 > 0
true
```

Variable Types

- `int, unsigned int`
- `bool`
- `float, double`
- `... // more to come`

Variable Types

- `int, unsigned int`
- `bool`
- `float, double`
- `... // more to come`

Do you know the “ranking” of these types when converting one to another?

Variable Types: Conversion Ranking

This is very important to keep in mind when writing complex expressions involving conversion of one of these (numeric) types into another:

```
bool < int < unsigned int < float < double
```

Variable Types: Conversion Ranking

This is very important to keep in mind when writing complex expressions involving conversion of one of these (numeric) types into another:

```
bool < int < unsigned int < float < double
```

IMPORTANT: `unsigned int` is “bigger” or more important than `int`, since it contains more possible positive values and is thus preferred in calculations.

For the “non-standard” types of `unsigned int` and `float`, there are suffixes to explicitly return these types in literals:

```
?? i = 3;    // int
?? j = 3u;   // unsigned int
?? k = 2.6;  // double
?? l = 2.6f; // float
```

Binary Representation: The Easy Part

Positional notation system, like in decimal and binary for integers, just continued beyond the decimal point:

binary	1	1	1	1	.	1	1	1
decimal	8	4	2	1	.	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$

What does $\texttt{101110.11001}_2$ translate to?

Binary Representation: The Easy Part

Positional notation system, like in decimal and binary for integers, just continued beyond the decimal point:

binary	1	1	1	1	.	1	1	1
decimal	8	4	2	1	.	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$

What does `101110.11001`$_2$ translate to?

$$2^5 + 2^3 + 2^2 + 2^1 + 2^{-1} + 2^{-2} + 2^{-5} = 48.78125_{10}$$

Easy, but tedious. We can simplify this to some degree if we have a C++ environment handy.

Binary Representation: The Easy Part (The C++ Way)

In C++ (i.e. also in [code]expert), we can simply input binary numbers with the prefix `0b`. The same goes for hexadecimal numbers which have the prefix `0x` and octal numbers which have the prefix `0`.

The following four lines all output 16:

```
std::cout << 16 << std::endl;      // decimal
std::cout << 0b10000 << std::endl; // binary
std::cout << 0x10 << std::endl;    // hexadecimal
std::cout << 020 << std::endl;     // octal
```

Sadly, this doesn't work directly for fractional numbers:

```
std::cout << 0b11.11 << std::endl;      // compiler error
std::cout << 0b1111 / 4.0 << std::endl; // 3.75
```

Now the other way around?

Binary Representation: The Tricky Part

Compute the binary expansions of the following decimal numbers.

1. 0.25
2. 11.1

Binary Representation: The Tricky Part - Solutions

1. $0.25_{10} = 0.01_2$

x	b_i	$x - b_i$	$2 \cdot (x - b_i)$
0.25	0	0.25	0.5
0.5	0	0.5	1
1	1	0	0

Binary Representation: The Tricky Part - Solutions

1. $0.25_{10} = 0.01_2$

x	b_i	$x - b_i$	$2 \cdot (x - b_i)$
0.25	0	0.25	0.5
0.5	0	0.5	1
1	1	0	0

2. $11.1_{10} = 1011.0\overline{0011}_2$
   11.1_{10} is first split into $11_{10} + 0.1_{10}$. The binary representation of 0.1_{10} is derived as follows:

x	b_i	$x - b_i$	$2 \cdot (x - b_i)$
0.1	0	0.1	0.2
0.2	0	0.2	0.4
0.4	0	0.4	0.8
0.8	0	0.8	1.6
1.6	1	0.6	1.2
1.2	1	0.2	0.4
0.4	0	0.4	0.8

   Hence, 11.1_{10} evaluates to $1011_2 + 0.0\overline{0011}_2$.

Normalized Floating Point Systems: Theory

What does the following set of numbers look like? Explain the individual parameters.

$$F^*(b, p, e_{\min}, e_{\max})$$

Normalized Floating Point Systems: Theory

What does the following set of numbers look like? Explain the individual parameters.

$$F^*(b, p, e_{\min}, e_{\max}) \\ = \{\pm b_0.b_1 b_2 \dots b_{p-1} \cdot b^e \mid b_i \in \{0, \dots, b-1\}, b_0 \neq 0, e \in \{e_{\min}, \dots, e_{\max}\}\}$$

b	base	2, 8, 10, 16, 60, ...
p	precision	24, 53, ...
$e_{\min}$	minimal exponent	-126, -1022, ...
$e_{\max}$	maximal exponent	127, 1023, ...

Special numbers like 0, ∞, ... are represented differently.

To try this for yourself:
`https://www.h-schmidt.net/FloatConverter/IEEE754.html`

Normalized Floating Point Systems: Exercise 1

Are the following numbers part of $F^*(2, 4, -2, 2)$?

1. 0
2. 2_{10}
3. $1.001_2 \cdot 2^{-1} \equiv 0.5625_{10}$
4. 0.53125_{10}
5. $1.111_2 \cdot 2^{-2} \equiv 0.46875_{10}$
6. 60_{10}

State the following numbers in $F^*(2, 4, -2, 2)$:

1. the largest number;
2. the smallest number;
3. the smallest non-negative number.

Compute how many numbers are in the set $F^*(2, 4, -2, 2)$.

Normalized Floating Point Systems: Exercise 1 - Solutions

Numbers 2, 3, 5 are part of $F^*(2, 4, -2, 2)$. 0 is represented differently, $1.0001_2 \cdot 2^{-1} \equiv 0.53125_{10}$ is too precise, 60 is too big.

Normalized Floating Point Systems: Exercise 1 - Solutions

Numbers 2, 3, 5 are part of $F^*(2, 4, -2, 2)$. 0 is represented differently, $1.0001_2 \cdot 2^{-1} \equiv 0.53125_{10}$ is too precise, 60 is too big.

1. The largest number is $1.111 \cdot 2^2$ which is 7.5 in decimal.
2. The smallest number is $-1.111 \cdot 2^2$ which is -7.5 in decimal.
3. The smallest non-negative number is $1.000 \cdot 2^{-2}$ which is 0.25 in decimal.

Normalized Floating Point Systems: Exercise 1 - Solutions

Numbers 2, 3, 5 are part of $F^*(2,4,-2,2)$. 0 is represented differently, $1.0001_2 \cdot 2^{-1} \equiv 0.53125_{10}$ is too precise, 60 is too big.

1. The largest number is $1.111 \cdot 2^2$ which is 7.5 in decimal.
2. The smallest number is $-1.111 \cdot 2^2$ which is -7.5 in decimal.
3. The smallest non-negative number is $1.000 \cdot 2^{-2}$ which is 0.25 in decimal.

The set has 80 numbers in it. This can be seen as follows. For a fixed exponent there are three digits we can vary freely, and for each number also the negative number is in the set, thus resulting in $2 \cdot 2^3 = 16$ numbers per exponent. On the other hand, there are 5 possible exponents, thus resulting in $5 \cdot 16 = 80$ numbers. Notice that in normalized number systems we cannot "count some numbers twice" as we've seen in the lecture that the representation of a number is unique.

Normalized Floating Point Systems: Exercise 2

Add $1.001 \cdot 2^{-1}$ (i.e. 0.5625) and $1.111 \cdot 2^{-2}$ (i.e. 0.46875) in $F^*(2, 4, -2, 2)$.

Normalized Floating Point Systems: Exercise 2 - Solutions

The two numbers $1.001 \cdot 2^{-1}$ (i.e. 0.5625) and $1.111 \cdot 2^{-2}$ (i.e. 0.46875) are added as follows:

1. Bring both to say exponent -1: $\quad 1.001 \cdot 2^{-1}$ and $0.1111 \cdot 2^{-1}$
2. Add as binary numbers:
$$\begin{array}{cr} & 1.001\ \ \cdot 2^{-1} \\ + & 0.1111 \cdot 2^{-1} \\ \hline & 10.0001 \cdot 2^{-1} \end{array}$$
3. Re-normalize: $1.00001 \cdot 2^{0}$
4. Round: $1.000 \cdot 2^{0}$

Notice that this does not coincide with their exact sum 1.03125 as the precision 4 is not high enough to represent this number.

Our Own System

Starting point: We want to build a 10bit system, as we're not as fast as computers yet.

We also want to imitate the scientific notation: 2.37×10^{12}
Thus we need 1 bit before the decimal point...

...and we take 5 bits after the decimal point, ...

...which leaves us with 4 bits for the exponent.

Our system so far:

$$F(2, 6, ?, ?)$$

What do we do with the 4 bits for the exponent?

Our Own System

We follow the IEEE 754 and take `unsigned int`, resulting in $e_{\text{min}} = 0, e_{\text{max}} = 15$, which is not that great, since we cannot represent numbers < 1.

Thus we introduce a "bias" of 8 to get: $e_{\text{min}} = 0 - 8 = -8, e_{\text{max}} = 15 - 8 = 7$

Our system now:

$$F(2, 6, -8, 7)$$

This is a solid starting point, but we can still improve this.

Our Own System, But Better

Now, we also want to have negative numbers, which requires an additional bit. Additionally, we want to follow the scientific notation more closely, by always writing a non-zero number before the decimal point.

This is perfect for us, since we can just exchange the meaning of the first bit to be the sign.

This translates to our system now being:

$$F^*(2, 6, -8, 7)$$

The * means that the bit before the decimal point is always one.

But there is one more thing to do...

Our Own System, Perfected to IEEE

We also want to be able to have some special values like 0, $\pm\infty$, NaN, …

We can do this by stealing a bit from the exponent (the negative side, i.e. $-8_{10} \cong 0000_2$) and using it for our special values like so, using the 5bit mantissa:

00000	± 0
00001	$\pm\infty$
00011	NaN

That is actually everything that is in the IEEE standard, there are no more special values, even though there is plenty of space.