Floating point numbers. The IEEE 754 standard.

In this article I will explain what floating-point numbers are with an overview of the IEEE 754 standard.

When dealing with numeric quantities, we can encounter integer numbers (without any fractional part) and numbers (real or floating-point) that consist both of an integer and a fractional part.

The first ones are easy to manipulate especially when their size in bits is less or equal than the parallelism level of the CPU (the number of bits a CPU can handle simultaneously).

The second category instead present some difficulties because the CPU does not know the meaning of decimal point and so, it is necessary some software support to correctly treat these quantities. Operations on real numbers take a long time to be executed (especially multiplications and divisions) and over the years, most chip makers, manufactured a series of dedicated chips, specialized in handling the floating-point numbers. Those chips are also known as FPUs (Floating Point Units).

The duty of these units that can be external to the CPU or included (built in) in it, is to perform floating-point operations in parallel to the CPU. Further since they must do only these tasks, they are highly optimized so that these operations are executed as fast as possible. This leads to a dramatic improvement of software performances.

Before diving in depth with floating-point numbers, let us give a look to an intermediate format called Fixed Point numbers.

If one wants to use numbers with fractional parts, but does not have the support of FPU and cannot use floating-point numbers software libraries, because they could slow down the performances (this is especially true with 8/16 bits micro controllers), it could fall back in the category of fixed numbers.

As the name says, they are quantities where a fixed number of bits is assigned both to the integer and to the fractional part.

Suppose to have 16 bits quantities (word), we could decide to assign 12 bits to the integer part and 4 bits to the fractional one (see figure below).

Bits 15 to 4	Bits 3 to 0
Integer part	Fractional part

Now we want to represent the number 100.375 using the fixed-point format.

Below the binary representation

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4
0	0	0	0	0	1	1	0	0	1	0	0	0	1	1	0

The 1st row shows the bit position (exponent) inside the word, the 2nd row shows the exponent of each bit (positive till bit 4), while the 3rd the value of each bit.

Note that for the fractional part (bits 3 to 0), the exponent is negative.

The integer part (100) is represented in binary as 000001100100 (remember we have 12 bits), while for the fractional part we have 0110, that means $1 * 2^{-2} + 1 * 2^{-3}$ = 0.25 + 0.125 = 0.375.

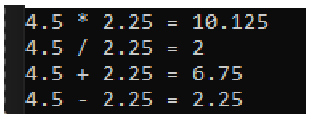
Following a simple C++ class that allows performing the basic four operations on fixe point numbers.

```
#include "pch.h"
#include <vector>
#include <string>
#include <sstream>
#include <iostream>

#pragma pack(1)

typedef struct
{
	union
	{
		struct
		{
			unsigned short fractional : 4;
			unsigned short integer : 12;
		};

		unsigned short number;
	};
} FixedPoint;

#pragma pack()

class FixedPointNumber
{
	FixedPoint fp;
	std::vector<std::string> fractParts { "0", "0625", "125", "1875",
	                                       "25", "3125", "375", "4375",
	                                       "5", "5625", "625", "6875",
	                                       "75", "8125", "875", "9375" };

public:
	FixedPointNumber() { fp.number = 0; }
	FixedPointNumber(unsigned short int integer, unsigned short  fractional)
	{
		fp.integer = integer;
		fp.fractional = fractional;
	}
	FixedPointNumber(FixedPointNumber const& other) { fp.number = other.fp.number; }

	FixedPointNumber& operator=(FixedPointNumber const& other)
	{
		fp.number = other.fp.number;
		return *this;
	}

	FixedPointNumber& operator+=(FixedPointNumber const& other)
	{
		*this = operator+(other);
		return *this;
	}
```

```
FixedPointNumber operator+(FixedPointNumber const& other)
{
	FixedPointNumber temp(*this);
	temp.fp.number += other.fp.number;

	return temp;
}

FixedPointNumber& operator-=(FixedPointNumber const& other)
{
	*this = operator-(other);
	return *this;
}

FixedPointNumber operator-(FixedPointNumber const& other)
{
	FixedPointNumber temp(*this);
	// Perform 2 complement of second operator
	unsigned short val = 1 + ~other.fp.number;
	// Then add the 2 operands
	temp.fp.number += val;

	return temp;
}

FixedPointNumber& operator*=(FixedPointNumber const& other)
{
	*this = operator*(other);
	return *this;
}

FixedPointNumber operator*(FixedPointNumber const& other)
{
	FixedPointNumber temp(*this);
	temp.fp.number *= other.fp.number;
	temp.fp.number >>= 4;

	return temp;
}

FixedPointNumber& operator/=(FixedPointNumber const& other)
{
	*this = operator/(other);
	return *this;
}

FixedPointNumber operator/(FixedPointNumber const& other)
{
	FixedPointNumber temp(*this);
	temp.fp.number <<= 4;
	temp.fp.number /= other.fp.number;

	return temp;
}
```

```
	std::string ToString()
	{
		std::stringstream ss;

		ss << fp.integer;
		if (fp.fractional)
		{
			ss << ".";
			ss << fractParts[fp.fractional];
		}

		return ss.str();
	}

	void PrintFixedPointNumber()
	{
		std::cout << ToString() << std::endl;
	}
};

int main()
{
	FixedPointNumber fixedPoint1(4, 8);
	FixedPointNumber fixedPoint2(2, 4);

	FixedPointNumber temp(fixedPoint1);
	temp *= fixedPoint2;
	std::cout << fixedPoint1.ToString() << " * " << fixedPoint2.ToString() << " = " <<
			temp.ToString() << std::endl;

	temp = fixedPoint1;
	temp /= fixedPoint2;
	std::cout << fixedPoint1.ToString() << " / " << fixedPoint2.ToString() << " = " <<
			temp.ToString() << std::endl;

	temp = fixedPoint1;
	temp += fixedPoint2;
	std::cout << fixedPoint1.ToString() << " + " << fixedPoint2.ToString() << " = " <<
			temp.ToString() << std::endl;

	temp = fixedPoint1;
	temp -= fixedPoint2;
	std::cout << fixedPoint1.ToString() << " - " << fixedPoint2.ToString() << " = " <<
			temp.ToString() << std::endl;

	std::cout << "Press Enter to terminate" << std::endl;

	int ch = getchar();
}
```

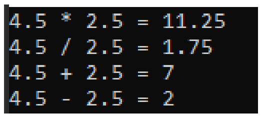
The figure below shows the output with two fixed point numbers initialized respectively with 4.5 and 2.25.



```
4.5 * 2.25 = 10.125
4.5 / 2.25 = 2
4.5 + 2.25 = 6.75
4.5 - 2.25 = 2.25
```

The limits of having fixed point numbers come when there are not enough bits to represent fractional parts that need more accuracy.

Consider the following output where the two fixed point numbers have been initialized respectively with 4.5 and 2.5.



```
4.5 * 2.5 = 11.25
4.5 / 2.5 = 1.75
4.5 + 2.5 = 7
4.5 - 2.5 = 2
```

Here we can see that the result of the division is wrong, it should be 1.8 instead of 1.75.

This is because we do not have enough bits on the fractional part to represent the quantity 0.05 (1.8 - 1.75).

Concluding, Fixed point numbers have the advantage to be easy to manage but, they are not versatile. In fact, if we would increase the precision, we must create a new type of fixed number by adding more bits to the fractional part against the integer one.

Doing this, we could increase the precision (decreasing the maximum allowed number in the integer part), but the new number will not be compatible with the previous one (the position of the decimal point has changed) and so no arithmetic operation is possible with these 2 numbers.

To overcome this limitation, floating-point numbers have been created.

As the name implies, floating-point numbers, have the possibility to move dynamically the decimal point, allowing the adjustment of the fractional part against the integer part (and vice versa) and giving the possibility to perform any arithmetic operation on number regardless of their decimal point position.

The IEEE 754 standard.