Queen Mary
University of London

ECS404U: COMPUTER SYSTEMS & NETWORKS

2020/21 – Semester 1

Prof. Edmund Robinson, Dr. Arman Khouzani

Lab Week 06: *Representing "Text" and "Real" Numbers (floating point) in Binary*

October 26/28, 2020

Deadline for submitting your proof of work: ***Next week's Thursday, at 10:00 AM UK time***

Student Name:

Yujeong JUNG

Student ID:

200699145

Brief Feedback to student (commendations, areas for improvement):

1 Text Representation in Binary

Learning Objectives

- Familiary with the general idea of how we represent characters in binary: by assigning each character a code-point number and representing that number in binary;
- Familiarity with non-unicode `ASCII`-based character encoding like `iso-latin-1` (officially `ISO-8859-1`), and unicode character encodings.
- Familiarity with the differences and similarities between implementation of unicode, namely, `UTF-8`, `UTF-16` and `UTF-32`.

1.1 overview

So far, we have practised how to represent numbers in Binary. We also learned why we are interested in binary system so much: because we can use electric potential (voltages) to represent zeros and ones, and then build circuits using transistors that can manipulate these 0s and 1s, and carry out logic and arithmetic operations on the numbers they represent.
But not all data is numbers, at least not until we represent them in numbers. One important example is "text" data. Because we only know how to build computers using transistors, and they only understand high and low voltages, we need to find a way to represent everything in binary, including text. But we should do that in a standard way that has the same meaning for everyone. You should remember from the lecture and course notes that this is why we have standard representations of characters (a.k.a., standard "encoding" of characters) like `ASCII`, `iso-latin-1`, and unicode ones like `UTF-8`, `UTF-16` and `UTF-32`. That is the focus of the first part of today's lab.

1. This question requires you to look at some short text files stored in different character sets in order to see the commonalities and differences at binary level.

   (a) There are some short text test files on QMPlus, prepared by saving the same text in `iso-latin-1`, `UTF-8` and `UTF-16`. Download these files and view them through the hex dump program. If you are using Linux or MacOs, the programme to display the raw content of a file is `xxd` in the terminal. If you are using Windows PowerShell, you can use `Format-Hex` programme.

   ```
   xxd test-text-iso-latin-1.txt
   ```

   For your convenience, we have provided the output that you would see in the following:

   ```
   0000000: 6162 6364 6566 4142 4344 4546 3031 3233  abcdefABCDEF0123
   0000010: 3435                                     45
   ```

   Also look at them as binary through `xxd -b`:

   ```
   xxd -b test-text-iso-latin-1.txt
   ```

   Here is the output you will see:

   ```
   0000000: 01100001 01100010 01100011 01100100 01100101 01100110  abcdef
   0000006: 01000001 01000010 01000011 01000100 01000101 01000110  ABCDEF
   000000c: 00110000 00110001 00110010 00110011 00110100 00110101  012345
   ```

   How are the letters represented in the different character sets?

   Check the characters against their `ascii` representation and verify that the binary and hex correspond. Each character should be eight binary bits and two hex digits. So in this example `01100010` corresponds to hex `62` and is character `b`.

   Note: you may see a weird character at the beginning of one of the files, with the hex representation of `feff`. This is a special unicode character called *Byte order mark* (BOM), which doesn't have a textual representation, but rather carries some information for the programmes reading the text: (https://en.wikipedia.org/wiki/Byte_order_mark).

   (b) View the content of the files `sample1.txt` and `sample2.txt`. You can use any text-viewer program, or simply from the Linux terminal, issue:

letter	lso-latin-1		utf-8		utf-16	
	hex	binary	hex	binary	hex	binary
a	61	01100001	61	01100001	61	01100001
A	41	01000001	41	01000001	41	01000001
e	65	01100101	65	01100101	65	01100101
E	45	01000101	45	01000101	45	01000101
0	30	00110000	30	00110000	30	00110000

`cat sample1.txt`

and

`cat sample2.txt`

Now, let's see how the file named `sample1.txt` is saved on the machine (by doing a "hex-dump"):

`xxd -p sample1.txt`

and

`xxd -p sample2.txt`

What do you see? Can you describe/explain the relation between the binary representations of these two files?

xxd -p sample1.txt: 6162 6364 6566 0a
xxd -p sample2.txt: 6162 6364 6566 4142 4344 4546 3031 3233 3435 0a5e 440a 2d64 0a

The binary of sample2 contains the binary array of sample1.
However, the difference is that sample 2 contains more numbers between the third and fourth numbers.

2 Floating point representation

Learning Objectives

- Familiarity with representation of floating point numbers in computers, technically, in the the IEEE 754 standard for double-precision floating-point numbers;
- Being able to convert real numbers from decimal to the standard binary representation and vice versa, given the standard format.

2.1 overview

In the previous labs, we quite extensively practised how computers represent unsigned (i.e. positive) and signed (positive or negative) integer numbers. We also learned how this can be used to represent other types of data as well, so long as we can represent any data with numbers, we can represent them. We saw that with the explicit example of the text data: if each characters is assigned a code number, then we are done: we already know how to represent numbers.

We didn't explore it explicitly, but you can already imagine this trick can be used to represent other types of data as well: images, audio, video, etc. can all be represented in binary, if we know how to represent them using numbers.

We also saw how dedicated hardware is built (from transistors!) to perform arithmetic and logic operations on these numbers.

However, all of the numbers we have dealt with so far were integers, i.e., whole numbers, e.g. -2, 130, etc. But what about numbers that are not whole, like 30.64, or -0.0237 (i.e., "real" numbers)? From week 5's lecture, you should recall that we do have a standard way to represent such numbers in binary as well, (documented in IEEE 754). Here, we will practice it through a few examples.

Note that you don't have to memorise the format necessarily. However, you should be able to work with the format (be able to convert back and forth between binary representation and their decimal equivalent) if the format is given to you.

2.2 Scientific Notation in base 10

Floating point representation of binaries (in IEEE standard) uses the *scientific notation* of representing real numbers. So let's make sure we understand that first. The scientific notation (in any base) just means that there should be only one non-zero digit before the "point". So the decimal number 453.02 is not in scientific notation, because it has 3 non-zero digits before the point. It can be expressed to scientific notation as $4.5302 * 10^2$. Here, 4.5302 is called the *mantissa*, and 2 is called the *exponent*. You can carry out the multiplication and confirm that the two numbers are equivalent!

2. Express the following decimal numbers with scientific notation. The first one is solved for you as an example:

real number (base 10)	in scientific notation (base 10)	mantissa	exponent
-153.02	$-1.5302 * 10^2$	-1.5302	2
0.012	1.2 * 10^-2	1.2	-2
2019.0	2.019*10^3	2.019	3
-0.0000405	-4.05*10^-5	-4.05	-5

2.3 Fixed point representation in base 2

First, let's interpret the number represented in binary using fixed point representation:

Going back to our example in base 10, we know that 453.02 really means the following:

$$453.02 = 4 * 10^2 + 5 * 10^1 + 3 * 10^0 + 0 * 10^{-1} + 2 * 10^{-2}$$

In particular, note that the numbers after the point just represent the coefficients of the negative powers of the base. This can be applied to any base, in particular, our favourite base of 2. So a number like 1001.01 in binary means:

$$1001.01 = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2}$$

Again, the digits after the point just represent the coefficient of the negative powers of the base. If someone was insisting to know what number this represents in decimal, we can simply answer it as:

$$(1001.01)_2 = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = 8 + 1 + 0.25 = (9.25)_{10}$$

Note that 2^{-2} is simply $\frac{1}{2^2} = \frac{1}{4}$ which is just 0.25 in base 10. Also note that of course in base 2 we can only have 0 or 1 as permitted digits!

3. Convert the following fixed point binary numbers to fixed point decimal (show your workings).

real number (base 2)	base 10
-1010.01	(-9.25)10
0.001	(0.125)10
1.11	(1.75)10
-11101.1101	(-29.8125)10

4. Convert the following fixed point decimal numbers to fixed point binary (show your workings).

   Example: 6.625. We have to find the coefficients of powers of 2 that can represent this number, including negative powers. We can do this by asking a series of questions. Obviously, we don't have to start with any power of 2 higher than 2 (why). So we proceed as follows:

$$\begin{aligned} 6.625 &= 1 * 2^2 + 2.625 \\ &= 1 * 2^2 + 1 * 2^1 + 0.625 \\ &= 1 * 2^2 + 1 * 2^1 + 1 * 2^{-1} + 0.125 \\ &= 1 * 2^2 + 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-3} \end{aligned}$$

   So we have: 6.625 in base 10 is equivalent to 110.101 in base 2.

   Now, your turn!

real number (base 10)	base 2
-0.25	-0.01
1.625	1.101
9.3125	1001.0101

2.4 Scientific notation in base 2

Remember the scientific notation: we are allowed to have a mantissa and exponent, the only rule is that the mantissa can have only one non-zero digit before the point.

So if we are given the binary number 110.101, its scientific notation expression will be $1.10101 * 2^2$. To see why, note that all we are doing is factoring out a 2^2:

$$(110.101)_2 = 1 * 2^2 + 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-3} = (1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-5}) * 2^2$$

5. Express the fixed point binary number in scientific notation:

real number (base2)	Scientific notation (base2)	mantissa (significand)	exponent
-1010.01	-1.01001*10^3	-1.01001	3
0.001	1*10^-3	1	-3
1.11	111*10^-2	111	-2
-11101.1101	-1.11011101*10^4	-1.11011101	4

6. Explain why the mantissa in binary always starts with 1 (Hint: the mantissa in any base should have only one non-zero digit before the floating point).

2.5 IEEE representation of floating points

We now have all the ingredients to represent floating point numbers using the IEEE standard.

Recall the format for double-precision (64 bit) representation of floating point numbers:

Now, using the answer to the previous question, find their representation of them in 64-bit IEEE format (which is also known as “double precision”). Just remember that since the mantissa in binary always starts with 1, we omit it (it will be a waste of space if we represent something that we always know what it is.) So the “fraction” is the mantissa but without the leading 1, so everything after the point.

Here is the detail:

- The first bit from the left is the “sign” bit, it will be 0 if the number is positive, and be 1 if the number is negative (so note that we do NOT use 2’s complement any more to represent negative floats).

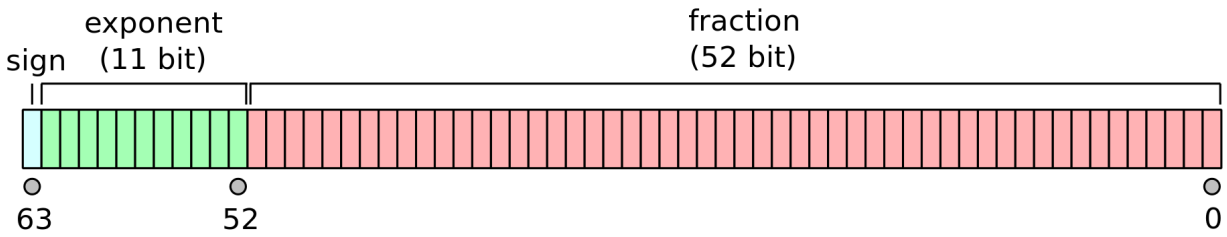

Figure 1: source: https://en.wikipedia.org/wiki/Double-precision_floating-point_format

- The next 11 bits represent the exponent, as follows: they represent the exponent you get in the scientific representation plus 1023 (which is called the "bias"). So the number that the exponent bits represent should be subtracted by the bias, 1023, to get the actual exponent. Note that this is also NOT two's complement way of representing singed numbers!
- The next 52 bits represent the fraction part of the mantissa. So the mantissa will be `1.fraction` where the fraction is these 52 bits. Again, we do this because mantissa in binary always has to start with 1, so we do not have to explicitly waste a bit for it![1]

So in short, the number represented is:

```
(-1)^{sign}  * 1.significand *  2^{exponent-1023}
```

Let's do an example. Remember that we worked out:

$$(6.625)_{10} = (110.101)_2 = 1.10101 * 2^2$$

Hence, to get its 64-bit IEEE floating point representation, we have:

sign bit 0 (because it is a positive number)

exponent bits exponent is 2, hence the exponent bits should represent 2+1023 =1025 (so that when we subtract 1023, we get the actual exponent, which is 2!). So the exponent bits should be 1025 in binary. Note that 1025 = 1024+1, so the 11 bits of the exponent will be `100 0000 0001`

fraction bits the mantissa (the significand) was `1.10101`. Recall that we do not represent the leading 1, so the fraction bits will be `10101`. The rest of the bits can be filled with zero, because 1.10101 is mathematically the same as 1.101010 or $1.101010\ldots0$ (why?).

So the overall bit sequence is:

`01000000 00011010 10000000 00000000 00000000 00000000 00000000 00000000`

or in Hex (so it is more compactly written), it is: `0x401A800000000000`

We can use this online tool to check our answer: http://www.binaryconvert.com/convert_double.html

Now your turn (these are from the previous parts). Derive the 64-bit floating point representation of the following decimal numbers:

7. (a) -10.25

base 2 representation: -1010.01

scientific representation in base 2: -1.01001*10^-3

sign bit: 1

exponent bits (11 bits): 10000000010

fraction bits (52 bits): 0100100000000000000

so the overall 64 bit sequence: 11000000 00100100 10 00000000 00000000 00

[1] There is a caveat to this rule through, when the exponent bits are all zeros (i.e., the exponent of -1023), then the interpretation of the mantissa is without the leading zero.

(b) 0.125

base 2 representation: 0.001
scientific representation in base 2: 1*10^-3
sign bit: 0
exponent bits (11 bits): 01111111100
fraction bits (52 bits): 0000000000000000000
so the overall bit sequence (64 bits) 00111111 11000000 00000000 00000000 00

2.6 Research Questions (optional)

Knowing what you learned about IEEE representations of floating points and a bit of online research, answer the following questions:

8. Fill out the following table:

	bit representation (in Hex)	in decimal
smallest positive number in 64-bit floating point?		
second smallest positive number in 64-bit floating point?		
largest positive number in 64-bit floating point?		
second largest positive number in 64-bit floating point?		

9. (a) What is the difference between the smallest positive number and the second smallest positive number in 64 bit floating point representation?

   (b) What is the difference between the largest positive number and the second largest positive number in 64 bit floating point representation?

   (c) Did you get the same number for the past two questions? What do you conclude about the precision of numbers in floating point representation?

10. (a) Explain why you should never use equality test in your codes when dealing with float numbers.

    (b) Given that you might have to do equality test in some algorithms, what should you do instead?

End of questions