

# MAT 3670: Lab 4

## IEEE Floating Point Representation

### Background

This lab continues the work from Lab 3, but now we focus our attention on the IEEE floating point standard.

### Python Hints

As you complete the functions for this lab, you may find the following Python features helpful:

- Strings  
`'MAT 3670'` — enclosed within quotes; act like lists in many ways. For example, if `s` is a string, `s[i]` gives the *i*th character.
- List slices  
`X[i : j]` — the sub-list consisting of `A[i]`, `A[i + 1]`, ..., `A[j - 1]`.
- List concatenation  
`A + B` — the list with items of `A` followed by items of `B`  
`32 * [0]` — 32 copies of `[0]`, joined by concatenation.
- List comparisons  
`X == [0, 0, 0, 0]` — true if and only if the two lists match.
- Exponentiation using floating point arithmetic  
`2.0**n` —  $2^n$  evaluated with floating point, not integer arithmetic (compare with `2**n`).
- Three-parameter `range` function  
`range(start, end, step)` — generate a list of values from `start` up to `end` by `step` size.

### Pre-Lab Exercises

1. To help verify work you will do for this lab, you will need to know some specific floating point values, expressed both as decimal and as 32-bit values. Complete Figure 1 before arriving at the lab. These conversions should be performed without computer and/or calculator assistance. (Work carefully, since it is easy to make a mistake.)

Binary	Hexadecimal	Decimal
1100 0000 1101 0100 0000 0000 0000 0000	C0D40000	-6.625
	41940000	
1100 0001 0001 0000 0000 0000 0000 0000		
0100 0000 0000 0000 0000 0000 0000 0000		
	C1900000	
		3,670.703125

Figure 1: Various floating point values, expressed in binary, hexadecimal, and decimal.

2. In many languages, it is legal to assign an integer quantity to a floating point variable. In order to make such an assignment, some internal conversions are required. Suppose  $k$  is a 32-bit signed, two's complement value which we wish to save as a 32-bit single-precision floating point value. For the following values of  $k$ , determine the appropriate 32 bits for the equivalent floating point value.

- $k = 12$  (or 0000000C)
- $k = -12$  (or FFFFFFF4)

3. During lab, you will be asked to implement the functions described in Figure 2. Good computing practice puts **thinking** before **coding**: your task prior to the lab is thus to thoroughly understand **what** each function is intended to do and **how** this can be achieved through a sequence of algorithmic steps. You can express these algorithmic ideas in any convenient form, such as pseudo-code, although these will ultimately need to be expressed in Python code.

- **hexToBinary( $s$ )**

This function is given a string  $s$  of hexadecimal digits and is to determine the corresponding list of binary digits. For example, `hexToBinary('5E7')` is `[0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1]`. Every digit of the input string contributes exactly four bits to the result. (*A solution for this function is given in lab4demo.py.*)

- **binaryToHex( $A$ )**

Given a list of binary digits  $A$ , this function yields the hexadecimal equivalent in the form of a string. For example, `binaryToHex([1, 1, 1, 0, 0, 1, 1, 1])` is `'E7'`.

- **singlePrecisionFloat( $A$ )**

Given a list  $A$  of 32 bits, this function yields the corresponding floating point value, using the (simplified) rules of the IEEE floating point standard. For example, `singlePrecisionFloat(hexToBinary('COD4000'))` is `-6.625`. As way of simplification, this function does not need to interpret the following categories of floating point values: denormalized,  $\pm\infty$ , and NaN.

- **signedIntegerToFloat( $A$ )**

Given a list  $A$  of 32 bits which represents a value in the two's complement system, this function produces a list of 32 bits which, when interpreted as an IEEE floating point value, matches the (integer) value corresponding to  $A$ . For example, if  $A$  is the list of bits corresponding to the integer value 12, then this function will produce the list of bits corresponding to the floating point value 12.0. If the value corresponding to  $A$  is too large, an approximation, obtained by "chopping" will be produced.

Figure 2: Functions to be implemented for Lab 4.

## Laboratory Exercises

1. Descend to your 3670 directory. Create a **lab4** directory to store the files you will create for this lab. Make a copy of your completed `lab3.py` and place it in your **lab4** directory, renaming it `lab4.py`.
2. Get a copy of the “demo” functions from the course website, placing it in your **lab4** directory.
3. Open `lab4.py` with **Aquamacs** and start the Python interpreter. Recall the need to import `lab4.py` and to reload it upon its modification.
4. Copy and paste the “demo” functions into your `lab4.py` file.
5. Implement each of the functions described in Figure 1. These functions can be completed and tested one at a time. After you have convinced yourself that your definition is correct, use the appropriate “demo” function to obtain a more thorough workout of your functions. These functions are summarized in Figure 3. Use a variety of test data, including the values you calculated for the pre-lab exercises.
6. Before you submit your work, be sure each of your functions is adequately documented with in-line comments.

<code>hexToBinaryDemo</code>	Generates all 2-digit hexadecimal quantities, invoking <code>hexToBinary</code> on each.
<code>binaryToHexDemo</code>	Generates all patterns of 8 bits, invoking <code>binaryToHex</code> on each.
<code>singlePrecisionFloatDemo</code>	Applies <code>singlePrecisionFloat</code> to each hexadecimal pattern in a given list.
<code>signedIntegerToFloatDemo</code>	Given a list of integers, converts each to a floating point value and prints a summary.

Figure 3: Demo functions for Lab 4.

## Submissions

When you have completed the lab, submit your `lab4` folder by dragging it onto the EIU submission icon.

## Contents of lab4demo.py

```
1  """
2  Python functions for MAT 3670, lab 4.
3  """
4  def hexToBinary(s):
5      """
6      Convert a hexadecimal string into its binary equivalent.
7      """
8      hexDigits = {'0': [0, 0, 0, 0],
9                  '1': [0, 0, 0, 1],
10                 '2': [0, 0, 1, 0],
11                 '3': [0, 0, 1, 1],
12                 '4': [0, 1, 0, 0],
13                 '5': [0, 1, 0, 1],
14                 '6': [0, 1, 1, 0],
15                 '7': [0, 1, 1, 1],
16                 '8': [1, 0, 0, 0],
17                 '9': [1, 0, 0, 1],
18                 'a': [1, 0, 1, 0], 'A': [1, 0, 1, 0],
19                 'b': [1, 0, 1, 1], 'B': [1, 0, 1, 1],
20                 'c': [1, 1, 0, 0], 'C': [1, 1, 0, 0],
21                 'd': [1, 1, 0, 1], 'D': [1, 1, 0, 1],
22                 'e': [1, 1, 1, 0], 'E': [1, 1, 1, 0],
23                 'f': [1, 1, 1, 1], 'F': [1, 1, 1, 1]}
24
25     # Build the result sequence of bits in 4-bit groups
26     result = []
27     for i in range(len(s)):
28         result = result + hexDigits[s[i]]
29
30     return result
31
32
33 def hexToBinaryDemo():
34     """
35     Generate all pairs of 2-digit hexadecimal strings and invoke hexToBinary on each.
36     """
37     hexDigits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F']
38     for MSB in hexDigits:
39         for LSB in hexDigits:
40             argument = MSB + LSB
41             print argument, hexToBinary(argument)
42
43
44 def binaryToHexDemo():
45     """
46     Generate all 8-bit patterns and invoke binaryToHex on each.
47     """
48     for k in range(2**8):
49         bitPattern = decimalToUnsigned(k, 8)
50         print bitPattern, binaryToHex(bitPattern)
51
52
53 def singlePrecisionFloatDemo(L):
54     """
55     Exercise the singlePrecisionFloat function by invoking it with each hexadecimal
```

```
56     pattern in the list L.
57     """
58     # Iterate over each element in L
59     for hexPattern in L:
60         print hexPattern, singlePrecisionFloat(hexToBinary(hexPattern))
61
62
63 def signedIntegerToFloatDemo(L):
64     """
65     Exercise the signedIntegerToFloat function by invoking it with each integer in L.
66     """
67     # Iterate over each element in L
68     for n in L:
69         # get the next value in L and convert it to a 32-bit, signed two's complement value
70         if n < 0:
71             nValue = twosComplement(decimalToUnsigned(-n, 32))
72         else:
73             nValue = decimalToUnsigned(n, 32)
74
75         # convert it to a floating point value
76         fValue = signedIntegerToFloat(nValue)
77
78         # obtain the decimal equivalent of this floating point value
79         dValue = singlePrecisionFloat(fValue)
80
81         # Show the values of interest
82         print binaryToHex(fValue), n, dValue
```