CMPE-240 Laboratory Exercise 05

IEEE Floating Point

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Daniel Santoro
Performed: 11/28/16
Submitted: 11/30/16

Lecture Section: 01
Professor: Alessandro Sarra
TA: Kevin Millar, Adrian Cruzat, Humza Syed

Abstract

The goal of this lab is to implement IEEE single precision floating point operations: encoding, multiplication, addition. Running on the Raspberry Pi, the program will run through tests for each operation with various variables and print the output to UART. The result was that the code executed and printed out the results of the computations correctly.

Design Methodology

The UART connection to the GPIO pins is set up (as shown in figure 1.1).

Figure 1.1

In the config.txt file of the Raspberry Pi image, UART needed to be enabled and Bluetooth needed to be disabled (as shown in figure 1.2).

```
# Enable UART
enable_uart=1

# Disable bluetooth (UART and bluetooth share the same resources,
# so both can't be active at the same time)
dtoverlay=pi3-disable-bt
```

Figure 1.2

Putty was used to connect over serial port via UART to the Pi. To find the COM number to connect to, device manager was used (on Windows 10) and the COM number was listed under Ports.

The result of the test cases was based on the following three tables (figure 1.4, 1.5, 1.6). To make outputting into a table format similar to table, a library for "printf" was modified to work with outputting via UART.

Expected encode/decode data:

FYI - Number	Integer (hex)	Fraction (hex)	IEEE (hex)
0.0	0x00000000	0x00000000	0x00000000
1.0	0x00000001	0x00000000	0x3f800000
0.5	0x00000000	0x80000000	0x3f000000
0.75	0x00000000	0xC0000000	0x3F400000
13.125	0x0000000D	0x20000000	0x41520000
0.8330078125	0x00000000	0xD5400000	0x3F554000
-13.125	0xFFFFFFF3	0x20000000	0xC1520000
8388608.0	0x00800000	0x00000000	0x4B000000

Figure 1.4

Addition data

IEEE (hex)	IEEE (hex)	IEEE Result (hex)	FYI Number
0x00000000	0x00000000	0x00000000	0.0+0.0 = 0.0
0x3f800000	0x3f800000	0x40000000	1.0+1.0 = 2.0
0x3f000000	0x3f000000	0x3f800000	0.5+0.5 = 1.0
0x41520000	0x41520000	0x41d20000	13.125+13.125= 26.25
0xc1520000	0xc1520000	0xc1d20000	-13.125+-13.125=-26.25
0x4B000000	0x3f000000	0x4B000000	8388608.0+0.5= 8388608.0* Note round off error

Figure 1.5

Multiplication data

IEEE (hex)	IEEE (hex)	IEEE Result (hex)	FYI Number
0x00000000	0x3f800000	0x00000000	0.0*1.0 = 0.0
0x3f800000	0x3f800000	0x3f800000	1.0*1.0 = 1.0
0x3f000000	0x3f000000	0x3e800000	0.5*0.5=0.25
0x41520000	0x41520000	0x432c4400	13.125*13.125 =172.26562
0xc1520000	0xc1520000	0x432c4400	-13.125*-13.125 =172.26562
0xc1520000	0x3f554000	0xc12eee80	-13.125*0.8330078125 = -10.9332275390625
0x4B000000	0x3f800000	0x4B000000	8388608.0*1.0 = 8388608.0

Figure 1.6

Before writing the code, the test cases were written in main. As each method was written, the corresponding test case was used to verify the results.

For the IeeeEncode function, it takes a struct number which has a “real” and “fraction” component. First check if the real number and fraction portion of the number or 0, return hex 0. Grab the sign of the real number. If negative, convert the real number to positive

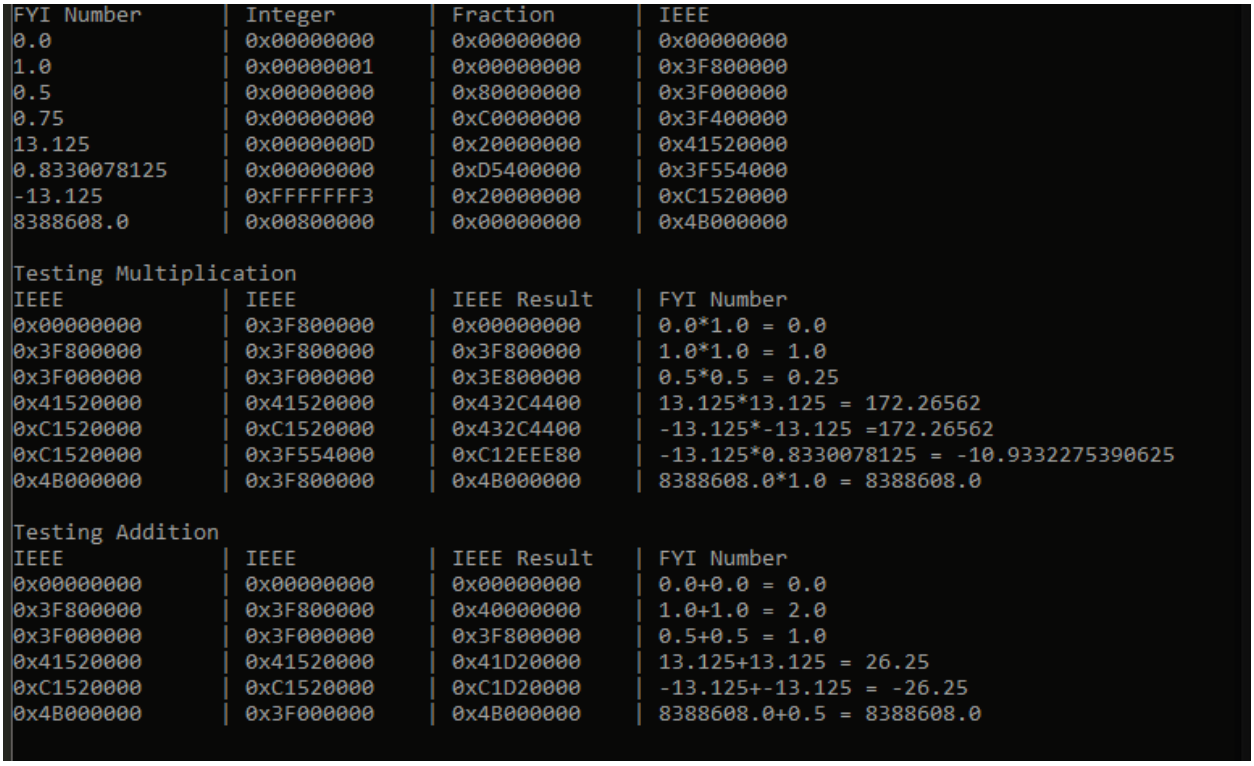
from two's compliment – flip the bits and add one. Shift the real number to the left and backfill from the high bit of the fraction. Shift the fraction one to the left. Continue until the 31st bit of the real number is 1. Keep track of the number of shifts. The mantissa is bits 0-22. Remove the implied 1. The bias is 158 – the number of shifts. The final number is the sign shifted into the 31st bit, the bias shifted into the next 8 bits, and the mantissa as the last.

For the IeeeMult function, it takes two floating point numbers. First check if either of the numbers is 0, and return 0. The sign is the xor of the two floating points numbers. Get the exponent by adding the two exponents and removing the redundant bias (127). The mantissa is the multiplication of the two mantissas with their implied 1's re-added and stored as a 64 bit number. The implied one was removed. The mantissa is normalized by shifting it right 8+16 bits and the exponent is increased by 1. The mantissa is cast into 32 bits. The final number is the sign shifted into the 31st bit, the bias shifted into the next 8 bits, and the mantissa as the last.

IeeeAdd takes two floating point numbers. Check if both the numbers are 0 and return 0. Check if the exponents are the same and notate that. If they aren't the same, make the smaller exponent equal to the higher one and shift the corresponding mantissa right until it is. The exponent is equal to either exponent since they are now the same. If only 1 sign is negative, the code will subtract the higher mantissa from the lower mantissa and the sign is equal to the sign of the higher mantissa's number (a larger negative number means the result is negative and vice versa). If the two mantissas are the same, return 0. If the signs are either both positive and negative, add the mantissas together. If negative, the sign is negative. If there was no shifting to make the exponent's equal, re-normalize the mantissa (shift right by 1) and increase the exponent by 1. Remove the implied 1. The returned number is the sign shifted into the 31st bit, the bias shifted into the next 8 bits, and the mantissa as the last.

Results and Analysis

The result of the lab was that the output over UART for the program matched the expected values of the given tables as shown in figure 2.1 (compared with figures 1.4, 1.5, 1.6).



```
FYI Number      | Integer       | Fraction      | IEEE
0.0             | 0x00000000    | 0x00000000    | 0x00000000
1.0             | 0x00000001    | 0x00000000    | 0x3F800000
0.5             | 0x00000000    | 0x80000000    | 0x3F000000
0.75            | 0x00000000    | 0xC0000000    | 0x3F400000
13.125          | 0x0000000D    | 0x20000000    | 0x41520000
0.8330078125    | 0x00000000    | 0xD5400000    | 0x3F554000
-13.125         | 0xFFFFFFF3    | 0x20000000    | 0xC1520000
8388608.0       | 0x00800000    | 0x00000000    | 0x4B000000

Testing Multiplication
IEEE            | IEEE          | IEEE Result   | FYI Number
0x00000000      | 0x3F800000    | 0x00000000    | 0.0*1.0 = 0.0
0x3F800000      | 0x3F800000    | 0x3F800000    | 1.0*1.0 = 1.0
0x3F000000      | 0x3F000000    | 0x3E800000    | 0.5*0.5 = 0.25
0x41520000      | 0x41520000    | 0x432C4400    | 13.125*13.125 = 172.26562
0xC1520000      | 0xC1520000    | 0x432C4400    | -13.125*-13.125 =172.26562
0xC1520000      | 0x3F554000    | 0xC12EEE80    | -13.125*0.8330078125 = -10.9332275390625
0x4B000000      | 0x3F800000    | 0x4B000000    | 8388608.0*1.0 = 8388608.0

Testing Addition
IEEE            | IEEE          | IEEE Result   | FYI Number
0x00000000      | 0x00000000    | 0x00000000    | 0.0+0.0 = 0.0
0x3F800000      | 0x3F800000    | 0x40000000    | 1.0+1.0 = 2.0
0x3F000000      | 0x3F000000    | 0x3F800000    | 0.5+0.5 = 1.0
0x41520000      | 0x41520000    | 0x41D20000    | 13.125+13.125 = 26.25
0xC1520000      | 0xC1520000    | 0xC1D20000    | -13.125+-13.125 = -26.25
0x4B000000      | 0x3F000000    | 0x4B000000    | 8388608.0+0.5 = 8388608.0
```

Figure 2.1

Conclusions

The lab was completed successfully. Major problems were encountered in each computation. Throughout the project, figuring out how to shift and mask bits caused the most struggles. In the encode function a major realization came from figuring out that negative numbers were represented in 2's compliment. In the add function, re-adding the implied 1 before multiplication, re-normalizing, and removing the implied 1. In add, finding edge cases and figuring out to handle addition of negative numbers. Also if there was a shift for the exponent, there wouldn't be a shift when re-normalization, had to keep track of weather the exponents were equal to begin with.