Introduction

In this lab, you will write and test a MIPS assembly language program to perform floating point addition. You should be familiar with the IEEE 754 Floating-Point Standard, which we have discussed in class and which is described in the text. Here we will be dealing only with positive single precision floating-point values, which are formatted as in Figure 1 (this is also described in Section 5.3.2 of your book).

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (8 bits)</th>
<th>Fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30 29 28 27 26 25 24 23</td>
<td>22 21 20 19 . . . 0</td>
</tr>
</tbody>
</table>

Figure 1: IEEE 754 Single-Precision Floating-Point Format

Remember that the exponent is biased by 127, which means that an exponent of zero is represented by 127 (01111111). (The exponent is not encoded using two’s complement.)

The mantissa is always positive, and the sign bit is kept separately. Note that the actual mantissa is 24 bits long: the first bit is always a 1 and thus does not need to be stored explicitly. This will be important to remember when you write your function!

There are several details of IEEE 754 that you will not have to worry about in this lab. For example, the exponents 00000000 and 11111111 are reserved for special purposes that are described in your book (representing zero, denormalized numbers, and NaN’s). Your addition function will only need to handle strictly positive numbers, and thus these exponents can be ignored. Also, you will not need to handle overflow and underflows.

To implement floating-point addition in assembly language, there are some MIPS instructions you will need to be familiar with.

- **and instruction:** First, you will need to make use of the and instruction to extract the exponent and fraction bits out of the floating-point numbers. This technique is called masking, because it involves the use of a 32-bit number that is used as a mask over another number to allow only certain bits of the result to be non-zero. For example, if you want to extract a number that is stored in bits 2-9 of a full 32-bit word held in $t1, you could use the following code:
addi $t1, $0, 0x625d

xtract: addi $t0, $0, 0x3fc  # create mask for all but bits 9:2
  and $t1,$t1,$t0        # extract bits 9:2 of $t1
  srl $t1,$t1,2         # shift right by 2 bits to read just
                      # the extracted bits

Figure 2. Example bit masking code

Observe that the hexadecimal mask 0x00003fc is the binary value 0000 0000 0000 0000
0011 1111 1100, a mask with bits 2 through 9 set to 1.

- or instruction: The or instruction is useful for combining bits.

- shift instructions: Other instructions that you will need to be familiar with are the shift
  instructions. You should take careful note of the difference between a logical right shift
  and an arithmetic right shift. The logical right shift simply shifts the bits right by the
  specified amount, always shifting in zeros from the left. The arithmetic right shift,
  however, keeps bit 31 the same and shifts all the other bits to the right, copying bit 31
  into all of the bits vacated by the shift. This allows negative two’s-complement numbers
  to be shifted right without changing their sign. There is another shift instruction available
  in MIPS that you should find quite useful: the variable shift. The instruction srav
  allows you to perform right shifts by a distance specified by the value in a register. As
  always, you should refer to Appendix B in your book for a complete reference on the
  MIPS instruction set.

Hand Analysis

Before implementing floating point addition, re-familiarize yourself with the representation
of floating point numbers and with carrying out addition by hand by answering the following
questions. Give your answers in binary and hexadecimal. For example, 1.0 is written as an
IEEE single-precision floating point number as:

\[
1.0 = 0 \ 01111111 \ 0000000000000000000000000 = 3F800000_{16}
\]

a) Write 2.0 as an IEEE single-precision floating point number.

b) Write 3.5 as an IEEE single-precision floating point number.

c) Write 0.50390625 as an IEEE single-precision floating point number.

d) Write 65535.6875 as an IEEE single-precision floating point number.

e) Compute the sum of the numbers from (c) and (d) and express the result in IEEE floating
   point format. Truncate the sum if necessary.

Writing the Floating Point Addition Program

You can add your floating-point addition function into the fpadd.asm file that is provided.
Carefully read the comments there, then add your code in the place indicated. Your code should
not modify any $s registers (because the program calling you expects that $s registers will not
change across procedure/function calls). You should be able to use only $t$ registers. Put the result of the addition in $v0$.

Notice that we have provided a number of masks (fmask, emask, etc.) at the top of the file for your convenience.

Your addition function need only handle strictly positive numbers, and need not detect overflow or underflow. Also, you need not perform rounding since it would be complicated and because truncation is also a valid option (although less accurate).

Your code will never actually need to perform a left shift for normalization, because of the restriction that it only needs to handle strictly positive numbers. Convince yourself that this is true before writing your code (think about the properties of the mantissas that get added and the properties that follow for the resulting sum).

Again, it is important to note that the most significant bit of the mantissa is an implied 1. After extracting the fraction bits by using masking, your code can use an or instruction to place the 1 back into the proper bit of the mantissas before performing addition on them. Having this implied 1 bit in place in the mantissas will make the normalization step more straightforward. Later, when your code reassembles a single floating-point value for its final result from a separate mantissa and exponent, you will need to remove this implied 1 bit from in front of the mantissa again.

Since your code will add only strictly positive numbers, the sign bits in the numbers being summed can be ignored. The sign bit of the resulting sum should be set to zero.

In summary, your algorithm will need to do the following:

1) Mask and shift down the two exponents.
2) Mask the two fractions and append leading 1’s to form the mantissas.
3) Compare the exponents by subtracting the smaller from the larger. Set the exponent of the result to be the larger of the exponents.
4) Right shift the mantissa of the smaller number by the difference between exponents to align the two mantissas.
5) Sum the mantissas.
6) Normalize the result. I.e., if the sum overflows, right shift by 1 and increment the exponent by 1.
7) Round the result (truncation is fine).
8) Strip the leading 1 off the resulting mantissa, and merge the sign, exponent, and fraction bits.

As a guideline: you should be able to implement the floating-point addition algorithm in under 50 lines of code; the solution uses 31 lines.
Testing your Program

Test your program on the following examples:

1.0 + 1.0 = 2.0 (0x3F800000 + 0x3F800000 = 0x40000000)
2.0 + 1.0 =
3.0 + 3.5 =
0.50390625 + 65535.6875 =

If your code is not working, don’t panic! You now have the opportunity to practice debugging assembly language code. Predict the result of each line of code for a known (and preferably simple) set of inputs. Step through the code one line at a time. Check that after each step, the results are what you expect they should be. When the results differ, you have found your bug. Correct your code, restart the simulation, and single step again until you verify that the correct answer is now produced.

What to Turn In

0. **Time:** Please indicate how many hours you spent on this lab. This will not affect your grade (unless entirely omitted), but will be helpful for calibrating the workload for next semester’s labs. This time should include the time for the entire lab, including pre-lab and post-lab.

**Pre-lab:**
1. Your answers to the floating-point hand analysis questions.

**Post-lab:**
2. A printout of your fpadd_xx.asm code (where “xx” are your initials) that shows your complete function that performs floating-point addition.
3. The results of your four addition test cases.