# Lab 7: Floating-Point Addition

## Introduction

In this lab, you will write a MIPS assembly language function that performs floating-point addition. You will then run your program using PCSpim (just as you did in Lab 6). For testing, you are provided a program that calls your function to compute the value of the mathematical constant *e*. For those with no assembly language experience, this will be a long lab, so plan your time accordingly.

## Background

You should be familiar with the IEEE 754 Floating-Point Standard, which is described in Section 3.6 of your book. (Hopefully you have read that section carefully!) Here we will be dealing only with *single precision* floating-point values, which are formatted as follows (this is also described in the "Floating-Point Representation" subsection of Section 3.6 in your book):

| Sign | Exponent (8 bits) | | | | | Significand (23 bits) | | | |
|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | ... | 24 | 23 | 22 | 21 | ... | 0 |

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 2s-complement. The significand is always positive, and the sign bit is kept separately. Note that the actual significand 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 NaNs). 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.

The program you are given to test your addition function estimates the value of the mathematical constant *e* by summing the first several terms in its infinite series, given below. It is the computation of this sum for which your floating-point addition function is needed.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

Fortunately, this series converges quickly, so the program will not have to compute very many terms. In fact, to compute the value of *e* to the precision available in IEEE 754 single precision numbers, we need to sum only about the first dozen or so terms.

To implement floating-point addition in assembly language, there are some MIPS instructions you will need to be familiar with. First, you will need to make use of the AND instruction to extract the exponent and significand 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 5–13 of a full 32-bit word, you could use the code below. Observe that the hexadecimal mask `0x00003FE0` is the binary value:

```
0000 0000 0000 0000 0011 1111 1110 0000
```

a mask with bits 5 through 13 set to `1`. Also, note that the mask label is an address in memory. The LW command is necessary to load the mask value from that memory address into a register before it can be used. For a command like `and $t1, $t1, mask` is illegal because the mask is not a register.

```
        .data
mask:   0x00003FE0      # mask for bits 5-13
        .text
        . . .
xtract: lw $t0, mask
```

```
and $t1,$t1,$t0
srl $t1,$t1,5
. . .
```

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, for example. 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 2s-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 the appendix Section A.10 on the CD for a complete reference on the MIPS instruction set.

## Hand Analysis

Before implementing floating point addition, 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 00000000000000000000000 = 3F80000016
```

- Write 2.0 as an IEEE single-precision floating point number.
- Write 3.5 as an IEEE single-precision floating point number.
- Write 0 as an IEEE single-precision floating point number.
- Write 0.3125 as an IEEE single-precision floating point number.
- Write 65535 as an IEEE single-precision floating point number.
- Compute 65535 + 0.3125 and express the result in IEEE floating point format.

## Writing the Code

You can add your floating-point addition function into the *flpadd.asm* file that is provided. This file already contains the code for a program that will compute *e* by calling your addition function, and can be found by clicking here.

Carefully read the comments there, then add your code in the place indicated. Remember that Spim is case-sensitive; use lower-case letters. Remember that your code should not modify any $s registers because the calling program calling expects that $s registers will not change across procedure calls. You should be able to use only $t registers to avoid saving and restoring $s registers on the stack.

The algorithm that you need to implement in your code is given in Figure 3.16 of your book. However, this algorithm can be simplified since your addition function need only handle strictly positive numbers and need not detect overflow or underflow. Also, you need not perform rounding (step 4 in the figure) since it would be complicated and because truncation is also a valid option (although less accurate). This leaves only steps 1–3 in the boxes at the top of the figure for you to implement.

With careful thought, you will also realize that in step 3 your code will never actually need to perform a left shift, 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 significands that get added in step 2 and the properties that follow for the resulting sum.)

Again, it is important to note that the most significant bit of the significand is an implied `1`. After extracting the significands by using masking, your code can use an OR instruction to place the `1` back into the proper bit of the significands before performing addition on them. Having this implied `1` bit in place in the significands will make the normalization step more straightforward. Later, when your code reassembles a single floating-point value for its final result from a separate significand and exponent, you will need to remove this implied `1` bit from in front of the significand 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 significands and append leading 1s
3. Compare the exponents and subtract the smaller from the larger
4. Set the exponent of the result to be the larger of the addend exponents
5. Right shift the significand of the smaller number by the difference between exponents to align the two significands
6. Sum the significands
7. If the sum overflows [1, 2), right shift by 1 and increment the exponent by 1
8. Strip the leading 1 off the sum significand
9. Merge the sum significand and exponent

As a guideline, you should be able to implement the floating-point addition algorithm in under 50 lines of code.

## Testing the Code

Before running your code, make sure that your PCSpim settings are as given from Lab 6. In testing your code, you may want to use it to simply add two predetermined numbers instead of to compute *e*. The comments in the code explain that you can do this by un-commenting the short segment of code that calls your addition function and commenting out the program that computes *e*. You can try adding simple numbers like 1+1 to debug your code first this way.

As you step through the code when you are running the program that computes *e*, compare the values that your procedure returns in $v0 with the values shown in bold in Figure 1. Having these values as a reference should help you verify that your addition function works correctly. You can also see from this table how quickly the series converges.

| Iteration | Sum of first i terms in series | | |
|---|---|---|---|
| | Ideal Decimal Value | Computed Hex Value | Computed Value in Binary |
| 0 | 1.0000000 | **3F800000** | 0 01111111 00000000000000000000000 |
| 1 | 2.0000000 | **40000000** | 0 10000000 00000000000000000000000 |
| 2 | 2.5000000 | **40200000** | 0 10000000 01000000000000000000000 |
| 3 | 2.6666666 | **402AAAAA** | 0 10000000 01010101010101010101010 |
| 4 | 2.7083333 | **402D5554** | 0 10000000 01011010101010101010100 |
| 5 | 2.7166666 | **402DDDDC** | 0 10000000 01011011101110111011100 |
| 6 | 2.7180555 | **402DF49D** | 0 10000000 01011011111010010011101 |
| 7 | 2.7182539 | **402DF7DD** | 0 10000000 01011011111011111011101 |
| 8 | 2.7182787 | **402DF845** | 0 10000000 01011011111100001000101 |
| 9 | 2.7182815 | **402DF850** | 0 10000000 01011011111100001010000 |
| 10 | 2.7182818 | **402DF851** | 0 10000000 01011011111100001010001 |
| ... | ... | ... | ... |

FIGURE 1 The correct return values for your program.

An explanation to eliminate confusion: Although your code should not make use of the floating point registers, the provided code uses them to compute *e*. While stepping through this code, be sure to examine the floating point single-precision registers, not the double-precision registers.

If your code is not working, don't panic! You now have the opportunity to learn to debug assembly language code. Step through the code one line at a time (using the F10 single step command in PCSpim). Check that after each step, the results are what you expect they should be. If you don't know what the result should be, take the time to figure it out; in the long run, it will save you debugging time. When the results differ, you have found your bug. Correct your code, open it again with PCSpim and single step again until you verify that the correct answer is now produced.

## Extra Credit

Add support for negative numbers to your function. This requires the following significant modifications, which could take approximately another 30 lines of code:

- Extracting the sign bits from the arguments and handling them appropriately
- Negating the significands for negative numbers before adding them
- Adding support for doing the proper number of left shifts (which will now be needed) in the normalization step
- Setting sign bit of result properly and negating significand of result when needed