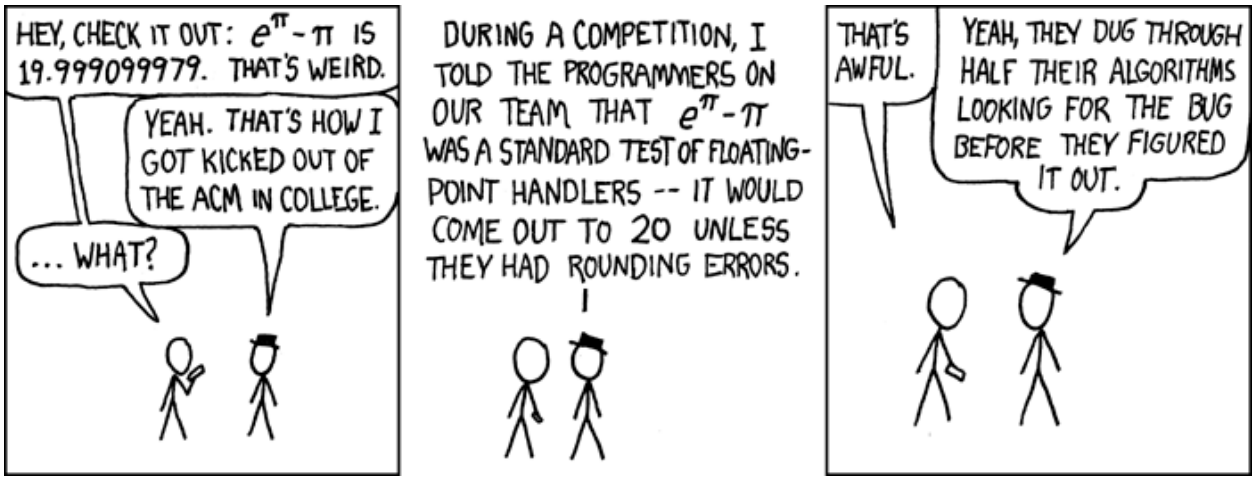
UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE012/L: COMPUTER SYSTEMS AND ASSEMBLY PROGRAMMING

LAB 6: FLOATING POINT MATH



HTTPS://WWW.XKCD.COM/217/

MINIMUM SUBMISSION REQUIREMENTS:

- Create a lab6 folder
- Lab6.asm in the lab6 folder
- README.txt in the lab6 folder
- Commit and Push your repo
- A Google form submitted with the Commit ID taken from your GITLAB web interface verifying that the above files are correctly named in their correct folders
- All of the above must be completed by the lab due date

LAB OBJECTIVE:

In this lab, you will learn to write subroutines (functions) and learn about IEEE single-precision floating point numbers and arithmetic.

As engineering projects become increasingly complex, it becomes necessary to develop some parts of the project independently. These parts must work together, even if they are not written by the same people or at the same time. It is crucial to carefully adhere to the specifications and standards that other pieces of code rely on. This is a key idea of software engineering, and is called *encapsulation*. That is, you can call code with no idea how it is written underneath as long as it adheres to the standard. In this lab, you will write code that interfaces with pieces of software that we have written (some of which you will not see).

LAB PREPARATION:

Read this document in its entirety, carefully.

Read *Introduction to MIPS Assembly Language Programming* by Charles Kann, Chapter 5. Note that Kann uses the word "subprograms" to refer to what we call "subroutines" or "functions."

© 2018, Computer Engineering Department, University of California Santa Cruz

Read about floating point numbers in Patt and Patel section 2.7.2 or in several available online resources (Google is your friend here!).

Learn about floating point values using MARS. It has an interactive tool to display floating point results under Tools->Floating Point Representation.

LAB SPECIFICATIONS:

In this lab, you will write a set of MIPS32 subroutines and some test code for those subroutines. These subroutines will perform various operations on floating point numbers without using the MIPS floating point instructions.

You may not use any MIPS instruction that has ".s" or ".d" in it. These include all instructions such as add.s, mul.s, abs.s, etc. **You may not use the mtc1 and mfc1 instructions either**. These use floating point hardware and this lab assumes we have a processor without a floating point unit. You may use these for debugging, but if you submit them in your code you will receive NO CREDIT. Note: You may want to use these in your test program! This is a convenient way to compute the results that your program should get. Make sure you do not include these instructions ANYWHERE (even in comments!) in Lab6.asm, however.

You may use integer MULT (or MUL) instructions. In fact, we encourage this for multiplying the mantissa when multiplying two FP numbers to get the mantissa product. Remember, the result of this will be a 64-bit value that then gets rounded to a 23-bit mantissa after normalization.

You will write the following four subroutines (though you are encouraged to write additional subroutines if you find it helpful):

```
# Subroutine PrintFloat
#     Prints the sign, mantissa, and exponent of a SP FP value.
# input:          $a0 = Single precision float
# Side effects:   None
# Notes:          See the example for the exact output format.

# Subroutine CompareFloats
#     Compares two floating point values A and B.
# input:          $a0 = Single precision float A
#                 $a1 = Single precision float B
# output:         $v0 = Comparison result
# Side effects:  None
# Notes:          Returns 1 if A>B, 0 if A==B, and -1 if A<B

# Subroutine AddFloats
#     Adds together two floating point values A and B.
# input:          $a0 = Single precision float A
#                 $a1 = Single precision float B
# output:         $v0 = Addition result A+B
# Side effects:   None
# Notes:          Returns the normalized FP result of A+B

# Subroutine MultFloats
#     Multiplies two floating point values A and B.
# input:          $a0 = Single precision float A
#                 $a1 = Single precision float B
# output:         $v0 = Multiplication result A*B
# Side effects:   None
# Notes:          Returns the normalized FP result of A*B

# Subroutine NormalizeFloat
#     Normalizes, rounds, and "packs" a floating point value.
# input:          $a0 = 1-bit Sign bit (right aligned)
#                 $a1 = [63:32] of Mantissa
#                 $a2 = [31:0] of Mantissa
#                 $a3 = 8-bit Biased Exponent (right aligned)
# output:         $v0 = Normalized FP result of $a0, $a1, $a2
# Side effects:   None
# Notes:          Returns the normalized FP value by adjusting the
#                 exponent and mantissa so that the 23-bit result
#                 mantissa has the leading 1(hidden bit). More than
#                 23-bits will be rounded. Two words are used to
#                 represent an 18-bit integer plus 46-bit fraction
#                 Mantissa for the MultFloats function. (HINT: This
#                 can be the output of the MULTU HI/LO registers!)
```

Your code should follow these additional specifications:

- Your code should work with our test_Lab6.asm without modifying it! You may create your own tests, but ensure it works with our version.
- Your subroutines should have no side effects besides those listed in the spec. In particular:
    - Only PrintFloat should do any syscalls.
    - External memory should NOT be altered unless described in the spec. (You may use the stack for function calls, however!)
    - All $s registers should have the same contents at the end of your subroutine as they held at the beginning. (Your code may reserve memory for its own use, if required)
- Your Lab6.asm file must be usable by another MIPS program that uses the directive ".include Lab6.asm".
    - Do NOT use the label "main:" in your file
    - Return properly from every subroutine
    - You must exactly use the following labels for the functions or you will receive no credit: "PrintFloat", "CompareFloats", "AddFloats", "MultFloats", and "NormalizeFloat".
- Your "AddFloats", "MultFloats" subroutines should call "NormalizeFloat".
- NormalizeFloat will take the sign bit, mantissa, and exponent and "pack" them into a normalized 32-bit float. The mantissa is up to 64-bits in two registers so it can be used with AddFloats and MultFloats. The mantissa has 18 integer bits and 46 fraction bits.
- CompareFloat will return a value of the comparison result.
- PrintFloat will print the sign, mantissa, and exponent.

You should not run Lab6.asm. Instead, we provide you with a script, test_Lab6.asm. This script should be placed in the same folder as Lab6.asm

Lab6.asm should only contain syscalls in PrintFloat to output the FP information and the comparison result.

As always, your code should be clean and readable. Refer to Lab4 to see the standard of readability we expect. In addition, we expect that your functions will have the headers containing their specifications (you may copy-and-paste from this document if you like).

We will test your code using a much more thorough battery of tests than contained in test_Lab6.asm. You should modify test_Lab6.asm to run your own tests, or generate your own testing code. You are encouraged to commit your own test code, but we will not check it as part of the grading process.

Here are some examples of the results of functions and what we expect:

PrintFloat with $a0=0x44801000:
Output:
SIGN: 0
EXPONENT: 10001001
MANTISSA: 00000000001000000000000

PrintFloat with $a0=0xBEDCFFFF:
Output:
SIGN: 1
EXPONENT: 01111101
MANTISSA: 10111001111111111111111

CompareFloats with $a0=0x44800000 $a1=0x3F000000:
Result in $v0: 0x00000001

CompareFloats with $a0=0x44800000 $a1=0x44800000:
Result in $v0: 0x00000000

CompareFloats with $a0=0x3F000000 $a1=0x44800000:
Result in $v0: 0xFFFFFFFF

AddFloats with $a0=0x44800000 $a1=0x3F000000:
Result in $v0: 0x44801000

AddFloats with $a0=0x3C0BBBBB $a1=0x3C111111:
Result in $v0: 0x3C8E6666

MultFloats with $a0=0x3B6BBBBB $a1=0xC2F00000:
Result in $v0: 0xBEDCFFFF

NormalizeFloat with $a0=0x00000000 $a1=0x00008080 $a2=0x00000000 $a3=0x00000086:
Result in $v0: 0x43808000

LAB STRATEGY:

As usual, an incremental development strategy is best for complicated projects. Here is a good strategy:

1. Start by figuring out how to use the test code. Generate some additional tests of your own.
2. Write pseudocode for each of these functions.
3. Write the PrintFloat function to handle so that you understand how to mask the portions of the floating point values.
4. Write the CompareFloats function so that you understand the representation of the floating point values.
5. Write the NormalizeFloat function so that you know how to put a value into normalized form.
6. Write the AddFloats function.
7. Write the MultFloats function.
8. Finally, write additional test code. Think outside the box. What happens with 0? What happens to the sign bit?

USING THE TEST CODE:

Subroutines must interact with other code. To emphasize this, we have written some code that will call your code: test_Lab6.asm. To use it, put it in the same folder as Lab6.asm, and run test_Lab6.asm in MARS. test_Lab6.asm calls the functions, so Lab6.asm must, at a minimum, contain those labels even if they aren't implemented yet. Start by just returning to implement an empty function.

Read the calling code carefully. It uses several MARS and MIPS techniques that you may be unfamiliar with. Ask questions if parts of this code do not make sense to you.

Notice that test_Lab6.asm contains a preprocessor macro at the very end, ".include Lab6.asm". This is how MARS includes your code in the assembly process. Upon assembly, you will see that your code has been "pasted" into the Text Segment, and that the Data Segment contains data from both Lab6.asm and test_Lab6.asm. (This is part of why you must be cautious about creating side effects in your code!)

As you progress in the lab, we expect you to generate your own test code. You may modify test_Lab6.asm as you wish. You may also make your own versions of the code inside it, and name them whatever you wish. Consider other approaches to writing test code that could simplify the testing process.

Testing is an extremely important skill in engineering, so treat this as an opportunity to develop your professional technique.

SUBROUTINES:

To interact correctly with the testing code, your code must use MIPS calling conventions. Information is passed into the subroutine using $a registers, and information is returned from the subroutine using the $v registers. The spec describes the specific role of each register. Furthermore, $s registers are meant to be preserved when subroutines are called, so their values are unchanged from the value it contained after the subroutine returns.

After a subroutine is complete, the $pc must point to the address of the instruction after the calling instruction (that is, the program counter should "return" to the calling code). Use "jal" to store the address of the return instruction in $ra, and "jr $ra" to return to the appropriate instruction.

Note that calling a subroutine inside of another subroutine will overwrite $ra, so you must "save" original values of $ra to return to the correct address. You should put the $ra and other saved register values on the stack.

ROUNDING, NAN, INF, AND -INF:

We will not check corner cases with NaN, INF and –INF. Because of this, we also do not care about "quiet" or "signaling" NaN if you read about that anywhere.

In some cases, you may have a number that has a repeating decimal. In this case, you should do your arithmetic with your mantissa to 32-bits precision for addition (use ADD) and 64-bit precision for multiply (use MULT). Digits beyond this should use "to nearest, ties to even" as discussed in lecture. Rounding will not be a big part of the grade if it is tested at all.

LAB WRITEUP:

Insert your Pseudocode that accurately reflects the structure of your code as a block comment at the top of your Lab6.asm file.

Please use our README.txt template on Canvas to make grading easier. Be sure your write-up (in the README.txt file) contains:

- Appropriate headers
- Describe what you learned, what was surprising, what worked well and what did not
- Answer the following questions:
    1. What additional test code did you write? Why?
    2. What is floating point overflow? Provide an example.
    3. Did you have any issues with rounding? How did you address them?
    4. Did you write any additional functions? What did they do?

To alleviate file format issues we want lab reports in plain text. Please ensure that your README.txt is a plain text file and CHECK IT ON GITLAB to make sure it is readable.

BUMPS AND ROAD HAZARDS:

Start this lab early. Ask questions.

Your success or failure in this lab will be determined to a large extend by the quality of your test code. The ability to run a variety of tests with little effort makes debugging much quicker. Do not treat your test code like an afterthought!

Subroutines may appear complex at first, but if you use them effectively, they actually reduce complexity. For example, calling PrintFloat or NormalizeFloat inside of AddFloats and MultFloats may seem like brain-twister, but the result is that you do not need to the printing and normalizing inside AddFloats and MultFloats. This way, you can split the debugging into smaller pieces. This can dramatically speed up the debugging process.

You may find it useful to create your own subroutines. This is encouraged! For example, a subroutine to split a FP number into the three components (but how would you **correctly** deal with more than two return values if only $v0 and $v1 can be used? HINT: stack frames.) Just make sure you use the appropriate headers for your subroutines so we understand what you did.