Machine Epsilon

In this program, we have two goals:

1. dust off our stale MIPS assembly programming skills and get back into programming shape
2. verify or debunk the rounding rules described in our text for double precision floating point

Machine epsilon is a term for the relative error due to rounding in floating point arithmetic. It is also sometimes called **unit roundoff**. This a description from the Octave help system for the `eps` command.

> More precisely, 'eps' is the relative spacing between any two adjacent numbers in the machine's floating point system. This number is obviously system dependent. On machines that support IEEE floating point arithmetic, 'eps' is approximately 2.2204e−16 for double precision and 1.1921e−07 for single precision.

Machine epsilon is sometimes described as the smallest positive number that can be added to 1.0 to give a result different from 1.0. Using our knowledge of the double precision floating point format, we showed in class that $1.0+2^{-52}$ can be represented exactly in double precision floating point, but $1.0 + 2^{-53}$ cannot.

Indeed, the value given in the Octave documentation for `eps` is

$$2^{-52} \approx 2.2204\text{e}{-16}.$$

We will see in the course of this program that there are smaller positive numbers that may be added to 1.0 to yield a result different from 1.0. To understand why, we need to remember how rounding is accomplished.

Our text tells us that internal calculations in the floating point unit include two extra bits of precision, the "round bit" and the "guard bit." Perhaps the most common rounding rule for these extra bits is:

00	round down
01	round down
10	round so the result has a 0 in its final digit
11	round up

With this rounding rule in mind, we can see that $1.0 + 2^{-53}$ rounds down to 1.0. But we should also be able to check that $1.0 + (2^{-53} + 2^{-54})$ should invoke the rounding-up rule, giving us a smaller positive number that can be added to 1.0 and increase it.

Requirements

We're going to check this with code. Ultimately, we are going to do several calculations, and your program will generate this sample output:

```
0.5^52 = 2.22044604925031308e-16
(1 + 0.5^52) - 1 = 2.22044604925031308e-16
(1 + 0.5^53) - 1 = 0
(1 + (0.5^53+0.5^54)) - 1 = 2.22044604925031308e-16
(1 + (0.5^53+0.5^105)) - 1 = 2.22044604925031308e-16
```

I want your solution to use functions and function calls. You should implement these functions (which I have listed from least complex to most complex).

1. printNewline() – This function requires no arguments. It should use the Spim syscall to print a string containing only a single newline character `"\n"`.
2. printString($a0) – This function requires a single argument $a0 containing the starting address of a NULL terminated string to be printed. It should also use the Spim syscall to print the string.
3. printDouble($a0) – This function requires a single argument $a0 containing the starting address of a double to be printed. It should load the value from data memory into register $f12 and use the Spim syscall for printing a double.

You will find information about the Spim system services on p. 88 of `MIPStextSMv11.pdf` available on Blackboard.

You should also implement these floating point functions.

4. addDouble($a0, $a1, $a2) – This function performs addition. The arguments $a1 and $a2 hold the addresses of doubles stored in data memory, and $a0 holds the address in data memory where the sum will be written. Your function should copy the data from data memory to floating point registers, do the addition, and copy the result back to data memory.

For example, if the data segment contained these definitions:

```
one: .double 1.0
half: .double 0.5
answer: .double 0.0
```

we could load $a0 with the address of answer, $a1 with the address of one, and $a2 with the address of half. A call to addDouble() would then store the value 1.5 in the address of answer.

5. subDouble($a0, $a1, $a2) – This function performs subtraction. The arguments $a1 and $a2 hold the addresses of doubles stored in data memory, and $a0 holds the address in data memory where the difference will be written. This function should be easy to write after you have perfected addDouble().

6. power($a0, $a1, $a2) – This function performs the calculation x^n where x is a double and n is an unsigned integer. The argument $a1 holds the address in data memory of the double x. The argument $a2 holds the unsigned value n. The argument $a0 holds the address in data memory where the result will be written.

We should be able to call power() using code similar to:

```
la $a0, answer
la $a1, half
li $a1, 52
jal power
```

Tips

Here are some things to think about that may help you develop your program with less effort.

- You won't be able copy directly between data memory and the floating point registers. You'll have to use integer registers as intermediaries (pairs of registers in fact, since double precision floating point numbers are 64-bit values).
- In the required example output, the text to the left of each equal (=) sign is just a string and can be stored in your program as a string literal.
- In the required example output, the numbers to the right of each equal (=) sign will be computed by code and printed with printDouble().
- You can store a string containing a single newline `"\n"` as a literal in your program.
- I recommend that you build and test one function at a time before moving on to the next function.
- Once all your functions work, you can structure the bulk of your program in your main section, making use of your helper functions. (Again, do each individual calculation and get it correct before moving on to the next.)

Final note: I view writing programs as *writing*, and I will grade you not only on the function of your program but also on the quality of your writing. In the context of programming, the quality of writing determines not only the correctness of execution, but also the clarity of your code.

Pay attention to how you order and arrange lines and sections of code, what registers you choose to make use of, where you skip lines and use white space, and the quality and detail of your comments. All of these things matter to me.

Conclusions

You should think about what the results of this code tell us about the rounding behavior actually used by the Spim simulator. Is this the same or different than the description given in the textbook? If this became an exam question on the next exam could you write an intelligent answer?