CS270 Programming Assignment 5
"LC3 Assembly Programming"

Program due Friday, November 18, 2011 (via Checkin by 2:59pm)

Assignments are to be completed individually. Discussing implementation specifics of the program is **forbidden**.

Goals

- Develop complex assembly programs with multiple subroutines.
- Write subroutines that follow calling conventions, and save/restore register state to/from the stack (i.e., use stack frames).
- Improve your understanding of the program stack.

The Assignment

In this assignment, you will be writing an LC-3 assembly program that adds two IEEE 16-bit floating-point numbers. The program takes two input LC-3 words **FLOAT_X** and **FLOAT_Y** that are in IEEE 16-bit floating-point format, adds these float values and returns the result in **FLOAT_SUM**. You must implement the program using multiple subroutines, and the subroutines must follow the conventions described below.

To learn more about the IEEE 16-bit floating point format, see the Wikipedia article at:
http://en.wikipedia.org/wiki/Half_precision_floating-point_format
To learn how to add two floating-point values, see the following article:
http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html

Register Allocation:

1. At any time in a subroutine, only registers R0 through R4 can be used to hold intermediate values. Registers R5 through R7 have special meaning and ***should not*** be used to hold intermediate values.
2. Register R5 is used to hold the current frame pointer.
3. Register R6 is used to hold the top of the stack (i.e. R6 is called the stack pointer).
4. Register R7 holds the return address of a calling function (i.e., linkage to caller).

The Stack:

The stack starts at memory location **0x4FFF** and grows towards lower addresses. That is, initially the top of the stack is at location 0x4FFF. Pushing a word onto the stack moves the top of stack to location 0x4FFE. ***Register R6 is used for pointing to top of the stack***.

Calling Convention (How parameters are passed to subroutines):

The caller pushes the last parameter that the subroutine accepts to the top of stack, and then pushes the next-to-last parameter on the stack, and so on until the first parameter of the subroutine is on top of the stack.

For example, if subroutine A calls subroutine B, and suppose B accepts two parameters x and y, then A pushes y on the stack first, and then pushes x on to the stack.

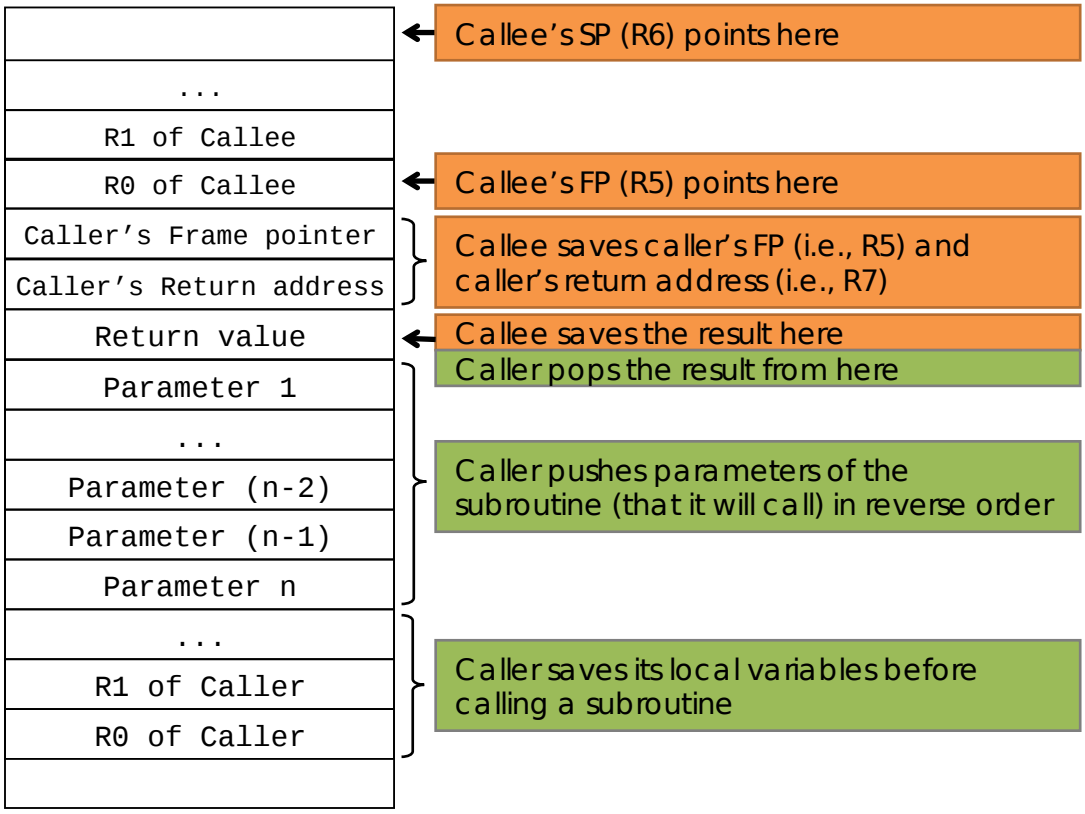
Return Value (How a subroutine returns a value to caller):

To return a value to the caller, the callee has to allocate a word on top the stack right above its parameters. The callee will save its result in this location and the caller will pop the result from the stack after the subroutine returns.

Caller-Save & Callee-Save:

1. Before making a subroutine call, the caller should save the contents of any registers it is actively using. For example if subroutine A is using R0 and R1, it should save those registers on top of the stack before making a call to another subroutine.
2. The caller saves the parameters of a subroutine on top of the stack in reverse order and then makes the call to the subroutine.
3. The callee should allocate a word on top of the stack (above the parameters) for return value.
4. The callee should save the contents of the register R7 on top of the stack.
5. The callee should save the contents of the register R5 on top of the stack.
6. The callee should save the contents of local registers it is currently using on top of the stack before making any subroutine calls.
    a. Note that the callee might need to save its registers if it is running out of registers to use for other operations (this is called register spilling).

Refer to the figure below to make sense of the above instructions.



Floating-point addition:

The program takes two IEEE 16-bit floating point numbers called FLOAT_X and FLOAT_Y. To add these numbers, follow this protocol:

1. Extract the exponent values from FLOAT_X and FLOAT_Y (these are main's two input variables); call these *exp_x* and *exp_y*.
    a. You might need to do some bit masking and right shifting for this purpose.
2. Compute the actual exponents from the extracted exponents; call these *ae_x* and *ae_y.*
3. Extract the fraction part from FLOAT_A and FLOAT_B; call these *frac_x* and *frac_y*.
    a. You might need to do some bit masking for this purpose (no shifting needed).
4. Add 0x0400 to both frac_y and *frac_x*
    a. This puts a 1 at bit-10 (note that fraction part has an implicit 1-bit).
5. Now, based on which actual exponent is smaller, select the appropriate fraction part
    a. For example, suppose *ae_x* is smaller, we need to select *frac_x*.
    b. If the exponents are the same, then you will need to determine if frac_a or frac_b has larger magnitude by comparing them directly.
6. If required, right shift the selected fraction by |*ae_x*– *ae_y*| (i.e., the absolute difference of the actual exponents) to align the significands (fractions).
7. Now that both fractions have the same exponent, and you know which fraction has a larger magnitude, add (or subtract) the fractions depending on their signs and magnitudes.
8. Re-normalize the fraction and compute the new exponent.
    a. You may need to right-shift and then mask out the implicit 1-bit.
9. Re-encode the resulting sign bit, exponent, and fraction into a new 16-bit FP number, and store this number in FLOAT_SUM.

Note that the above protocol leaves out a few special cases.

1. What if the actual exponents are equal?
    a. Nothings gets right shifted, but the resulting fraction may need to be re-normalized
    b. ***You MUST handle this case***.
2. What if original exponent value is either 0 or 31 (i.e., *exp_a* or *exp_b* is either 0 or 31)?
    a. ***You do not need to handle this case.***

You are free to name your subroutines with whatever naming conventions you would like to follow. We provided you with skeleton code and several of the functions (from previous assignment) filled in. These functions follow all the rules described above (i.e., use stack frames). Read these functions carefully and try to understand their structure. This will help you with your assignment.
The skeleton code can be downloaded from here:
http://www.cs.colostate.edu/~cs270/Assignments/PA5/floatadd.asm

Submission Instructions

1. Name your assembly file floatadd.asm and store it in a directory called PA5.
2. Submit a tar.gz file generated from the following command (assuming you are running this command from inside PA5 directory):
```
$ cd ..; tar –czvf PA5.tar.gz PA5
```

Do not submit the assignment with lc3tools inside PA5. You may lose points if you do so.

Grading Criteria

Points will be awarded for your main routine writing the correct floating point number to the address FLOAT_SUM (output variable), given a FLOAT_X and FLOAT_Y (your input variables) as follows:

- Correct FLOAT_SUM for x and y, same sign, different exponents: 25 points
- Correct FLOAT_SUM for x and y, different signs, different exponents: 25 points
- Correct FLOAT_SUM for x and y, same signs, same exponents: 20 points
- Correct FLOAT_SUM for x and y, different signs, same exponents: 20 points
- Following submission instructions and code style: 10 points

Late Policy

Late assignments will be accepted up to 48 hours past the due date, but with 10% deduction for every 24 hours late. Assignments will not be accepted past this period. If you were unable to submit via Checkin for any reason, contact us via email and we may be able to help.