Computer Systems Organization
CSCI-UA.0201-005 Fall 2024

Programming Assignment 2
Due Wednesday, October 23 at 11:59pm

In this assignment, you will be implementing integer and floating point arithmetic by manipulating bits, as well as doing some simple assembly-language programming.

To complete the assignment, perform the following steps.

1. Download one of the following files attached to this assignment, according to the system you are using:
   - assignment2_macos.tgz for macOS
   - assignment2_linux.tgz for Linux
   - assignment2_cygwin.tgz for Windows/Cygwin

   Save and uncompress the downloaded file in the directory where you want to work on and compile your program. To uncompress the file, in a shell, type

   tar -xzvf *filename*

   where *filename* is the name of the file that you downloaded.

   The four files that are extracted from the compressed file are:

   - main.c, a C file that contains main().
   - exponent.s, an assembly language file that you will be adding code to.
   - makefile, a file that makes compiling quick and easy (do not change this file).
   - Hints.pdf, a file containing helpful hints.

2. Create a new file called **functions.c** (please spell it exactly as stated, no capitalization). At the top of that file put the usual **#include** line to allow you to call printf. Then:
   - In functions.c, write a function **print_bits** that takes an integer x as a parameter, doesn't return anything, and prints out the individual bits of x. Your code can look like the code I wrote in class in Lecture 10 (see "Resources->Programs Discussed in Class" on Brightspace), except that instead of shifting x, you must shift the mask used to check each bit of x. Rather than assuming that an int is 32 bits (as I did in my code), you should use sizeof(int).
   - Create a new file called **functions.h** and put the prototype of print_bits into functions.h, so that print_bits can be called from other files. As you may remember, a function prototype contains only the return type, the function name, and the parameter list for the function (followed by a semicolon).

- In main.c, uncomment the code labeled as "Section 1" in the main() function, so that print_bits is called.
- To compile and test your print_bits code, start a shell and, in the directory where your code is, type "make". This will use the makefile I've supplied you to compile the the code and create an executable called assign2 (or assign2.exe on Windows). To run the compiled code, type "./assign2". The program will ask you to enter an integer and then will call your print_bits function.
- Once you are satisfied that your print_bits code works, move on to the next step.

3. In functions.c, write a function int_multiply that takes two 32-bit signed integers, x and y, as parameters and returns a 64-bit signed integer – namely a **long** – as the result (since product of two 32-bit numbers can require 64 bits). It should return the result of multiplying x and y, but without using the built-in multiplication operator *. Instead, it should perform integer multiplication by using integer addition and shifts according to the algorithm that I discussed in class. Your code should do the following:
   - Copy the values of x and y into signed 64-bit integer variables lx and ly.
   - Declare a 64-bit signed integer variable to hold the result and initialize it zero.
   - Write a loop that iterates over the bits of ly, starting at the rightmost bit and going to the leftmost bit (i.e. bit 63), such that each time through the loop:
     - using a mask to determine the value of the current bit of ly, if the current bit of ly is 1, add lx to the result.
     - shift lx to the left by one bit.

     Be sure to use a 64-bit mask. You can either declare a 64-bit variable that you use as the mask or you can shift "(long) 1" to the left by the appropriate number of bits (the "(long)" cast tells the compiler to treat 1 as a 64-bit number).

   When you have finished writing int_multiply, put the prototype for it in functions.h and uncomment the code of main() in main.c labeled "Section 2". Then, compile the code using "make" and run it as specified above. Please note that you must use shifting, you cannot simply add x together y times.

4. In functions.c, write a function float_multiply that takes two 32-bit floating point numbers a and b (i.e. of type **float**) and returns a 32-bit floating number resulting from multiplying a by b. You will not be using the built-in floating point multiplication operator, *, but instead will be performing the multiplication by manipulating the bits as I discussed in class. Your code should do the following:
   - Copy the bits of a and b into 32-bit unsigned integer variables val_a and val_b. You cannot just do an assignment, but rather you'll need cast the addresses of a and b to type (unsigned int *) and then dereference the addresses. See the lecture notes from Lecture 12.
   - If val_a or val_b are zero (i.e. all the bits are zero), then return 0.0. Otherwise, continue with the rest of the steps.
   - Extract the exponent and mantissa fields from val_a and val_b, storing each field into its own unsigned integer variable. Be sure that the exponent and mantissa values are shifted to the rightmost position in their respective variables. That is,

the 8-bit exponent should be at bit positions 0 through 7 of the variable storing the exponent and the 23-bit mantissa should be at bit positions 0 through 22 of the variable storing the mantissa. Note that at this point, each exponent value still contains the bias (127) and the mantissa value is missing the implicit 1 before the point.

- Compute the exponent of the result, which is the sum of the exponents from val_a and val_b, minus one bias (as discussed in class in Lecture 12).
- Compute the mantissa of the result by doing the following:
  - Insert the leading 1 into each mantissa value from val_a and val_b. Note that this 1 goes at bit position 23.
  - Declare a 64-bit unsigned integer variable (i.e. of type **unsigned long**) and write the result of calling your int_multiply function on the mantissa values from val_a and val_b (with the 1 inserted in the previous step).
  - Because multiplying two numbers that each has one one non-zero bit before the point and 23 bits after the point will give a result that has either one or two non-zero digits before the point and 46 bits after the point, we need to adjust the result as follows:
    - shift the 64-bit variable from the previous step to the right by 23 bits. This will give us a mantissa value that has the desired 23 bits after the point.
    - If there are two non-zero bits before the point, i.e. if bit 24 of the 64-bit variable is 1, then renormalize by shifting the 64-bit variable to the right by one bit and incrementing the exponent of the result (see above) by one.
  - Copy the value of the 64-bit mantissa variable into a 32-bit unsigned int variable. No shifting is needed in this step.
- Extract the sign bits from val_a and val_b (moving them to bit position 0) and compute the sign of the result. Notice that the sign of the result of a multiplication is simply the result of the XOR operation on the signs of the operands.
- Finally, construct the result of the multiplication by inserting into a 32-bit unsigned integer variable the computed sign bit, the computed exponent bits, and the computed mantissa bits (just the lower 23-bits, masking out the explicit leading 1 at bit position 23). Return the 32-bit variable as a float by using the trick discussed in Lecture 12 of casting the address of the variable to type (float *) and then dereferencing the address.

When you have completed the float_multiply function, put its prototype in functions.h and uncomment the code labeled “Section 3” in the main() function in main.c. Then compile and run your code, as specified above, to test and debug it.

5. In the file exponent.s, fill in the missing assembly code for a function, exponent, that takes three 32-bit integers, x, y and n, and returns a 32-bit integer containing the value of x^n+y^n. The code and the comments in the file should be self-explanatory. When you have filled in the missing assembly code, create a file exponent.h and put a C prototype

for the exponent function in that file. Then, uncomment the code labeled “Section 4” in the main() function in main.c and compile and test your code using the same process as above.

6. When your code is working perfectly, upload the files functions.c, functions.h, exponent.s, and exponent.h to Brightspace (nothing else). As always, do <u>not</u> submit programming assignments that don’t work perfectly. The late penalty is much less than the penalty for code that doesn’t work.