PHYS 3437: Computational Methods in Physics, Assignment 1

Set January 14th due Jan 29th

Q1. (a) What do the numbers 00000100_2 and 11111100_2 represent in two's complement notation? Calculate their product using binary multiplication and show your working. (**2 marks**)

(b) If $b = b_0 b_1 ... b_{n-1}$ is the n-bit representation of the integer x then we can write x as a sum over the digits of b as follows:

$$x = \sum_k b_k 2^{n-1-k}.$$

I have deliberately left off the limits on k - you need to figure them out. The exponent on the 2 involves minus signs because the left most b_k corresponds to b_0 which must multiply 2^{n-1}. Two's complement representations are derived from the notion of arithmetic on a repeating number line. As was discussed in the lecture, the value of 2^n is mapped back to zero (although the system has effectively overflowed in this case), *i.e.* $1111....1111_2+1=10000....0000_2$. For n-bits of precision, so that the length of the number line is 2^n, we call the resulting arithmetic system 2^n *cyclic arithmetic*.

Show using proof by induction (*i.e.* show that the result is true for $n = 1$, and if it assumed true for $n = k$ then it can be shown to be true for $n = k + 1$) that

$$2^n = 1 + \sum_{i=0}^{n-1} 2^i$$

(**2 marks**).
(c) The above result is the first step in showing why addition of numbers defined using two's complement leads to 2^n cyclic arithmetic. The next step is to give a mathematical function equivalent to the NOT operator, *i.e.* define an $f(b)$ (for any bit b) such that $f(1) = 0$ and $f(0) = 1$. This can be achieved by a very simple operator - what is it? (**1 mark**).
(d) Using the above results show that for any input number x, by applying the operations involved in defining two's complement (the NOT and then add 1) we derive a number that is equivalent to $-x$ under 2^n cyclic arithmetic. (**4 marks**)

Q2. (a) As we discussed in the lecture, floating point arithmetic is not associative, so that $(A+B)+C \neq A+(B+C)$. Demonstrate this using A=1234.565, B=45.68044, C=0.0003 (you may assume, for simplicity, a decimal floating-point system that carries 7 significant figures, and ensure you use appropriate rounding at each step of the calculations). Also test whether these particular values are distributive or not, *i.e.* if $(A + B) \times C = A \times C + B \times C$ and again show all the steps. (**2 marks**)

(b) What is the largest *integer* that can be represented *exactly* in the IEEE 754 floating-point standard at single precision? By "exactly" I mean a number that if you subtract 1 from it, you will still get the correct answer (*i.e.* there will be no rounding). What about at double precision? (HINT: Think about the spacing of floating point numbers on a number line, also how much precision do you have in the mantissa? Note, the answer isn't simply the largest number that you can write in single precision floating point format.) (**2 marks**)

The IEEE 754 normalization step is usually discussed slightly differently from the method we examined in class. The fraction part $f = 0.d_1d_2...d_t$ is defined by

$$m \times 2^{e+1}/2^t = \pm 2^e(1+f)$$

where we require $f < 1$ as our normalization. The number of bits set aside for storage in single precision will be 1 bit for the sign, 8 bits for the exponent and 23 bits for the fractional part (32 bits total). The logical layout of the bits is

[sign bit][bits of exponent][bits of fraction]

Given this normalization assumption, we can apply the following steps to determine the floating point representation of a decimal number x, in IEEE 754:

- The sign bit is given by 1 if x is negative, zero otherwise
- You can find the exponent, e, by successively testing values of e until $x/2^e = (1+f) = 1.d_1d_2...d_t$
- Once the value of e is found the binary representation will be biased with 127 (*i.e.* the stored value will be the binary representation of e-127)
- Finding the fractional part is a matter of summing different powers of 2^{-n}, until you match the decimal value. The logical storage order is left to right with the leftmost digit correspond to 2^{-1}.

(c) Work out the IEEE 754 single precision representation of -0.1328125_{10}. What is this in hexadecimal? Give both the big endian and little endian representations. (**4 marks**)

Q3. Download the source code `prime.f` from the class website. This is the same as that documented in the FORTRAN primer, except I have removed the comments. Open the file in an editor and add comments to the source code to make it appropriately well commented. You don't have to copy the comments in the primer, rather you can use a style that you are happy with and feel is suitably personal. Your goal though in commenting is to make the code clear for *another* person to read, as well as yourself. Submit a print out of your code with your assignment (**2 marks**).

Once you have commented the code you can compile the code with `f77 prime.f -o prime`. This will create an executable named `prime`, and you can run it by just typing `prime`. Try it for a few numbers and make sure it works.

a) There is a small bug in the program - what is it? How do you fix it? (**2 marks**)
b) How many prime numbers are there less than two million? (**1 mark)**
c) What is the 20th largest prime number less than one million? (**1 mark**)

Q4. The program `prime.f` is not optimal in its initial form. Firstly, since we know 2 is the only even prime number the main program doesn't need to check the even integers. Secondly, the subroutine `divisor` doesn't need to check even divisors, since again the only even prime number is 2. Modify `prime.f` to avoid these unnecessary computations, but ensure that the number 2 is still included in your list of output primes, and also included in the `icount` variable.

As well as submitting a hardcopy of your revised code in your assignment, email me a copy. You will be graded on style, whether the code works, and whether you have made the algorithm faster with your changes. (**4 marks**)