This lab considers loss of significance in the operation of subtraction of two close floating point numbers as a consequence of the hardware implementation of floating point numbers. Specifically, we have the following result for the error involved in $f_l(f_l(x) - f_l(y))$:

If $x$ and $y$ are two normalized floating point numbers with $x > y > 0$ and

$$2^{-p} \leq \left(1 - \frac{y}{x}\right) \leq 2^{-q}$$

for some positive integers $p$ and $q$, then at most $p$ and at least $q$ significant binary bits are lost in the subtraction $x - y$. For an example of a direct application of this theorem look at the function `lost_sigdigits()` in the program `subtraction_thm.c`. In this program we stop the loop after we have lost one-half (26) of our binary significant digits.

1. Get copies of the sample program `sigdigits2.c` and the makefile (`Makefile`) in one of your subdirectories. Compile and link the program by typing:

```
gcc -g -lm sigdigits2.c -o sigdigits < cr >
```

or

```
make sigdigits2 < cr >
```

2. Try running the program (you just need to type the name of the executable, `sigdigits2`). Compare the first output with the second. Make sure you go through the algebra and trigonometric identities to check that the limit

$$\lim_{h \to 0} \frac{\sin(x)\sin^2(h)}{(1 + \cos(h))h} + \frac{\cos(x)\sin(h)}{h}$$

is the same as the limit above. The expression has simply been manipulated so that there are no longer subtractions involving two very close double-precision floating point numbers. Hence, our results are now accurate.

3. Get copies of the sample program `floatrep.c`, the hardware debugging routines `hardware.c`, and the makefile (`Makefile`) in one of your subdirectories. Compile and link the program by typing:

```
gcc -g floatrep.c hardware.c -o floatrep < cr >
```

or

```
make floatrep < cr >
```

This program uses the auxilliary module `hardware.c` and it will show you important ma-chine parameters for single and double precision floating point operations on the machine you run it. You can also use the routines in `hardware.c` to dump floating point numbers in binary, octal, or hexadecimal.

**Assignment** As before, there are two parts to this assignment. The first part is to write a program to estimate the limit

$$\lim_{h \to 0} \frac{\sqrt{1+h} - 1}{h}$$
not by doing it directly but by algebraically manipulating the expression in order to eliminate the subtraction of two very close double-precision floating point numbers. Email me the path to your program in plain text. For example, you might say that the path to your solution is

/usr/stu/yourusername/cs305/lab2.c

This is all I need. Do not email attachments of code to me.

The second part is to answer the following questions. You can either turn in your answers on paper or email me your answers in a plain text file.

1. Why should the correct answer of the above limit be the value of the derivative of $\sqrt{x}$ at $x = 1$?

2. Suppose that $x \geq y > 0$ and the expression $x + y$ is computed (in floating point) as $\text{fl}(\text{fl}(x) + \text{fl}(y))$. What is an upper bound on the relative error (in terms of the machine epsilon)? Suppose that the addition in the expression is replaced by subtraction?

3. The code block below finds the area of a ring, or annulus, with minor radius $r$ and major radius $R$. Would you want do change anything? Why or why not?

```c
float pi = 4.0*atan(1.0);
double area, r = 0.973020621, R = 1.000300521;
area = pi*(R + r)*(R - r);
```

4. The designers who wrote the IEEE floating point specification allowed for 53 binary significant digits for a `double` (counting the leading “1” which is understood plus the 52 digit mantissa). What would have been the consequences if they actually took 60 digits for the mantissa (instead of 52)?

5. Suppose that $x$ and $y$ are double precision numbers which are very close to one another as in the code below. Mathematically, $z$ and $w$ should be equal (using the properties of the logarithm) but will that be the case when the code is run? Why or why not?

```c
double x = 3.0000000000000005;
double y = 3.0000000000000000;
double z, w;
z = log(x) - log(y);
w = log(x/y);
```