Of course, we know that the solutions of the equation \( ax^2 + bx + c = 0 \) can be found by using the formula

\[
\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

And, we can also determine that there are no real solutions of the equation if the value of the discriminant is negative:

\[ b^2 - 4ac \]

Consider implementing a MIPS program to solve quadratic equations:
- it should allow the user to specify the coefficients
- the coefficients and solutions will be decimal values (not integers)
- the solver should detect the case there are no solutions

This is a good excuse to examine the floating-point facilities available for MIPS programmers.
MIPS includes two coprocessors that support specialized execution features. One is dedicated to computations involving floating-point values:

- 32 registers to store 32-bit (*single precision*) floating-point values in IEEE 754 format
- support for "coupling" pairs of registers to store 64-bit (*double precision*) floating-point values in IEEE 754 format
- a floating-point ALU
MIPS assembly includes an impressive collection of instructions for performing computations with floating-point values; the latest release of SPIM supports most of them.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.float</td>
<td>Directives for declaring 32- and 64-bit floating-point data</td>
</tr>
<tr>
<td>.double</td>
<td></td>
</tr>
<tr>
<td>lwcl</td>
<td>Load single from memory to FP register (aka l.s)</td>
</tr>
<tr>
<td>ldc1</td>
<td>Load double from memory to FP register (note the pattern)</td>
</tr>
<tr>
<td>swcl</td>
<td>Store single from FP register to memory (aka s.s)</td>
</tr>
<tr>
<td>sdc1</td>
<td>Store double from FP register to memory</td>
</tr>
<tr>
<td>add.[s</td>
<td>d]</td>
</tr>
<tr>
<td>mul.[s</td>
<td>d]</td>
</tr>
<tr>
<td>sub.[s</td>
<td>d]</td>
</tr>
<tr>
<td>div.[s</td>
<td>d]</td>
</tr>
<tr>
<td>neg.[s</td>
<td>d]</td>
</tr>
<tr>
<td>sqrt.[s</td>
<td>d]</td>
</tr>
</tbody>
</table>
Selected MIPS Floating-Point Instructions

There are also many instructions for conversion, comparison and branching. Here is a sampling of the single-precision instructions; each has a double-precision analog.

- `cvt.s.d` convert double to single
- `cvt.s.w` convert fixed-point or integer to single
- `c.eq.s` set coprocessor flag according to result of comparison
- `c.lt.s`
- `c.le.s`

- `bc1[t|f]` branch to address if coprocessor flag is true|false

The MIPS Architecture for Programmers, Volume II is an excellent reference for the complete MIPS instruction set.
A few things are obvious:
- the solver must receive the three coefficients as parameters
- the solver must be able to indicate to the caller that there are no solutions
- otherwise, the solver must return two solutions (possibly equal) to the caller

The following implementation is based upon these decisions:
- the coefficients will be passed via the stack
- the error code will be communicated via register $v0
- the solutions, if any, will be placed into the registers $f11 and $f12
- the solver will use single-precision numbers

When the solver begins execution, the stack will be in the following logical state:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
  sp
```

...
Quadratic Solver Call

```assembly
main:
    addi $sp, $sp, -12 # reserve stack space for the # coefficients
    la $a0, prmpt1 # get coefficient of x^2
    jal get_coefficient
    s.s $f0, 8($sp) # put it on the stack

    la $a0, prmpt2 # get coefficient of x
    jal get_coefficient
    s.s $f0, 4($sp) # put it on the stack

    la $a0, prmpt3 # get constant term
    jal get_coefficient
    s.s $f0, 0($sp) # put it on the stack

    jal quad_solver # call the quadratic eq'n solver

    beq $v0, $zero, OK # check exit code from solver
```
Quadratic Solver

```assembly
quad_solver:
   .data
   two: .word 2
   four: .word 4
   .text
   l.s $f0, 8($sp)                # retrieve coeffs from stack
   l.s $f1, 4($sp)
   l.s $f2, 0($sp)
   li $v0, 0                    # default to success
   # calculate discriminant
   mul.s $f8, $f1, $f1           # f8 = B^2
   mul.s $f9, $f0, $f2           # f9 = A*C
   l.s $f5, four
   cvt.s.w $f5, $f5                # f5 = 4.0
   mul.s $f9, $f9, $f5           # f9 = 4*A*C
   # test discriminant
   c.lt.s $f8, $f9                # is B^2 < 4*A*C?
   bclf isOK                    # if not, compute solutions
   li $v0, 1                    #    else, set error code
   jr $ra                       #    and quit

   # ... continues ...
```
Quadratic Solver

```assembly
# OK, compute solutions
isOk:
     neg.s  $f9, $f9                  # f9 = -4*A*C
     add.s  $f9, $f8, $f9             # f9 = B^2 - 4*A*C
     sqrt.s $f9, $f9                  # f9 = sqrt(B^2 - 4*A*C)
     mov.s  $f7, $f1
     neg.s  $f7, $f7                  # f7 = -B
     l.s    $f5, two
     cvt.s.w $f5, $f5
     mul.s  $f8, $f5, $f0             # f8 = 2*A
     add.s  $f10, $f7, $f9             # f10 = one root
     div.s  $f10, $f10, $f8
     neg.s  $f9, $f9
     add.s  $f11, $f7, $f9             # f11 = other root
     div.s  $f11, $f11, $f8
     jr     $ra

######################################################################## end quad_solver
```
get_coefficient

# Prompt the user to enter an integer value. Read and return
# it. It takes the address of the prompt as its only parameter.

get_coefficient:

li $v0, 4  # system call code for printing a string = 4
syscall   # call operating system to perform
          # print operation

li $v0, 6  # system call code for reading a float = 6
syscall   # system waits for input, puts the
          # value in $f0

jr $ra

end get_coefficient
So far we have only considered *leaf procedures*, that is, procedures that do not make calls themselves.

Non-leaf procedures must save their return address before executing a `jal`, and then restore that value before executing a return. For example:

```assembly
get_quadratic:
    addi $sp, $sp, -4        # preserve return address on the stack
    sw $ra, ($sp)

    la $a0, prmpt1          # get coefficient of x^2
    jal get_coefficient
    s.s $f0, 12($sp)        # put it on the stack

    # ... get and save the other coefficients

    lw $ra, ($sp)           # restore the return address
    addi $sp, $sp, 4
    jr $ra

end get_quadratic
```