An Example of A MIPS Program Using Procedures and Parameters

/*
 * This module implements a procedure (solve) that computes the roots of a
 * quadratic equation that has integer roots, returning them to the caller.
 * The arguments are the coefficients of the quadratic equation (input) plus
 * the two roots (output). It also returns a status code to the caller:
 * 
 * 0 - Computation successful and root values are valid
 * 1 - Roots are not integers (roots values are truncated)
 * 2 - Roots are complex (root values invalid)
 * 3 - Overflow occurred during computation (root values invalid)
 * 
 * Register usage:
 * 
 * Parameters:  
 *  $4 = A (by value)  
 *  $5 = B (by value)  
 *  $6 = C (by value)  
 *  $7 = address to receive first root  
 *  $8 = address to receive second root  
 * 
 * Return value:$2  
 * 
 * Temporaries: $2, $3  
 * 
 * *** This version of the program does not incorporate overflow handling
 * *** code. It will crash if overflow occurs in computing the discriminant.
 * 
 * R. Bjork - 2/99
 * 
 */

# The .section assembler directive is used to break a program into
# sections. Executable code goes in the .text section.

.section .text

# Each procedure needs to have its entry point declared as a label; if
# it is called from outside this module its entry point must also be
# declared as a global symbol (for the linker). The name should
# also be declared by a .ent directive (for the debugger).

.ent    solve
.globl  solve

solve:

# Upon entry, a non-leaf procedure must allocate a frame on the
# stack, and save its parameters and return address, as well as any
# callee-saved registers it intends to use. (None in this case)
# The frame may also be used to hold local variables. (None in this
# case) The size of the frame must be a multiple of 16
# The .frame and .mask directives provides information for the debugger
# about the structure of the frame.
# The first argument of .frame indicates what register is used to point
# to the frame (either the stack pointer or some other register set
# aside for that purpose); the second gives the size of the frame, and
# the third argument indicates what register holds the return address
# for the procedure (almost always $31).

.frame  $sp, 32, $31
# The mask directive specifies what registers are saved in the stack frame, and where the register save area begins relative to the start of the frame. The first argument is a bit mask with 1's in bit positions corresponding to registers that are saved. Only registers in the callee saved set ($16 and up) normally appear in the mask. (The only register this procedure needs to save in this group is the return address - $31). The second argument indicates the offset from the high end of the frame ($sp + size) to the slot where the highest numbered register specified in the mask is saved. In this case, $31 is saved 24 prior to the high end of the frame, so the offset is -24.

```
mask 0x80000000, -24
```

# The code that follows actually creates the frame and saves the registers in it.
```
addi $sp, -32
sw $31, 8($sp)
sw $4, 12($sp)
sw $5, 16($sp)
sw $6, 20($sp)
sw $7, 24($sp)
sw $8, 28($sp)
```

/* Compute the discriminant (put in $2). Registers already contain * the correct parameters */
```
jal compute_discr
```

/* Test for negative discriminant */
```
slt $3, $2, $0
beq $3, $0, d_ok # Non-negative, so go on
addi $2, $0, 2 # Status value for complex roots
b fini # Exit
```

d-ok:

/* Compute square root of discriminant (put in $2) */
```
add $4, $2, $0 # Put discriminant in $4 as parameter
jal compute_sqrt # $2 now contains sqrt(discriminant)
```

/* Compute the roots */
```
lw $4, 12($sp) # First parameter = A
lw $5, 16($sp) # Second parameter = B
add $6, $0, $2 # Third parameter = sqrt(discriminant)
jal compute_roots # $2 and $3 now contain the roots
```

/* Save the roots in location specified by caller */
```
lw $7, 24($sp) # Restore return parameter addresses
lw $8, 28($sp)
sw $2, 0($7) # Store first root
sw $3, 0($8) # Store second root
```
/* Check to be sure they are integers - if not, status code will
* indicate that a warning about truncation is needed.
*/
lw $4, 12($sp)  # First parameter = A
lw $5, 16($sp)  # Second parameter = B
lw $6, 20($sp)  # Third parameter = C
add $7, $2, $0  # Fourth parameter = first root
add $8, $3, $0  # Fifth parameter = second root
jal test_roots   # $2 contains 0 if roots OK, 1 if not

/* Exit protocol for solve. When this point is reached, $2 must
* contain the status code to be returned to the caller
*/
#
# Upon exit, a non-leaf procedure must restore its return address and
# any callee-saved registers from the stack frame and then deallocate
# the frame. (The parameters need not be restored).
fini:
lw $31, 8($sp)
addi $sp, 32

# Return to caller
jr $31

# Each procedure must end with a .end directive
.end solve

/*
* The following local routine computes the discriminant.
*
* Parameters: $4 = A
* $5 = B
* $6 = C
* Return value: $2
*/
#
# As a local routine, its name does not need to be declared global, and
# as a leaf routine, it does not need to save anything on the stack.
# A frame directive with a size of 0 indicates no frame.
.ent compute_discr
  .frame $sp, 0, $31

compute_discr:
mulo$2, $5, $5  # Pseudoinstruction. Assembler generates code to
      # put 32-bit product in $2; check for overflow and
      # raise an exception if one occurs. #2 = B*B
addi $3, $0, 4  # $3 = 4
mulo$3, $3, $4  # $3 = 4*A - overflow checked
mulo$3, $3, $6  # $3 = 4*AC - overflow checked
sub $2, $2, $3  # $2 = B*B-4AC = discriminant - overflow checked
jr $31

.end compute_discr
The following local routine computes the integer square root of the discriminant.

Parameter: $4 = \text{discriminant}$
Return value: $2 = \text{integer square root (truncated if need be)}$

Method: Successive testing of individual bits, starting with $2^{15}$ and working down to $2^0$

```
compute_sqrt:
    ; guess at square root 0 - initially 0
    add $2, $0, $0
    ; bit mask for trial bit
    ori $3, $0, 0x8000

sqrt_loop:
    ; or in trial bit
    or $2, $2, $3
    ; test to see if guess is now too big
    mul $5, $2, $2
    slt $5, $4, $5
    beq $5, $0, bit_ok
    ; set trial bit back to 0
    xor $2, $2, $3
    ; move on to next bit
    srl $3, $3, 1
    bne $3, $0, sqrt_loop
    ; set trial bit back to 0
    ; notify next level
    jr $31

compute_root:
    ; $4 = 2*A
    add $4, $4, $4
    ; $5 = -B - overflow checked
    sub $5, $0, $5
    ; $2 = -B - sqrt(discriminant) - overflow checked
    sub $2, $5, $6
    ; $2 = first root
    div $2, $2, $4
    ; $2 = second root
    add $3, $5, $6
    div $3, $3, $4

    jr $31
```
/* 
* The following local routine tests the roots to be sure they are 
* integers
* 
* Parameters: 
* $4 = A$ 
* $5 = B$ 
* $6 = C$ 
* $7 = first root$ 
* $8 = second root$ 
* 
* Return value: $2 = 0$ if roots are integers, $1$ if not 
* 
* Method - verify that $A * sum of roots = -B$, $A * product = C$ 
* 
*/

test_roots:
    .ent test_roots
    .frame $sp, 0, $31

    test_roots:
        add $2, $7, $8    # $2 = sum of roots
        mul $2, $2, $4    # $2 = A * sum of roots
        add $2, $2, $5    # $2 will be 0 iff A*sum of roots = -B
        bne $2, $0, not_int
        mul $2, $7, $8    # $2 = product of roots
        mul $2, $2, $4    # $2 = A * product of roots
        sub $2, $2, $6    # $2 will be 0 iff A*prod of roots = C
        bne $2, $0, not_int
        jr $31            # Return with $2 = 0 - roots OK

    not_int:
        addi $2, $0, 1
        jr $31            # Return with $2 = 1 - roots not OK

.end test_roots