Procedure Support

From previous study of high-level languages, we know the basic issues:
- declaration: header, body, local variables
- call and return
- parameters of various types, with or without type checking, and a return value
- nesting and recursion

At the machine language level, there is generally little if any explicit support for procedures. This is especially true for RISC architectures.

There are, however, many conventions at the assembly language level.

Procedure Call and Return

Calling a procedure requires transferring execution to a different part of the code… in other words, a branch or jump operation:

\[
\text{jal} \quad \text{<address>} \quad \# \quad \text{$ra = PC + 4} \\
\quad \# \quad \text{PC = <address>}
\]

MIPS reserves register $31, aka $ra, to store the return address.

The called procedure must place the return value (if any) somewhere from which the caller can retrieve it. The convention is that registers $v0 and $v1 can be used to hold the return value. We will discuss what to do if the return value exceeds 4 bytes later…

Returning from the procedure requires transferring execution to the return address the \text{jal} instruction placed in $ra:

\[
\text{jr} \quad \text{$ra} \quad \# \quad \text{PC = $ra}
\]
Passing Parameters

In most cases, passing parameters is straightforward, following the MIPS convention:

```
$a0     # 1st parameter
$a1     # 2nd parameter
$a2     # 3rd parameter
$a3     # 4th parameter
```

The called procedure can then access the parameters by following the same convention.

What if a parameter needs to be passed by reference? Simply place the address of the relevant data object in the appropriate register, and design the called procedure to treat that register value accordingly.

What if a parameter is smaller than a word? Clever register manipulation in the callee.

What if there are more than four parameters? We'll discuss that later…

MIPS Memory Organization

In addition to memory for static data and the program text (machine code), MIPS provides space for the run-time stack (data local to procedures, etc.) and for dynamically-allocated data:

```
Stack

Dynamic data

Static data

Text

Reserved
```

Dynamic data is accessed via pointers held by the program being executed, with addresses returned by the memory allocator in the underlying operating system.
The System Stack

MIPS provides a special register, $sp, which holds the address of the most recently allocated word on a stack that user programs can employ to hold various values:

Note that this run-time stack is "upside-down". That is, $sp, decreases when a value is added to the stack and increases when a value is removed.

So, you decrement the stack pointer by 4 when pushing a new value onto the stack and increment it by 4 when popping a value off of the stack.

Using the System Stack

MIPS programs use the runtime stack to hold:
- "extra" parameters to be passed to a called procedure
- register values that need to be preserved during the execution of a called procedure and restored after the return
- saved procedure return address, if necessary
- local arrays and structures, if any
System Stack Conventions

By convention, the caller will use:
- registers $s0 - $s7 for values it expects to be preserved across any procedure calls it makes
- registers $t0 - $t9 for values it does not expect to be preserved

It is the responsibility of the called procedure to make sure that if it uses any of the registers $s0 - $s7 it backs them up on the system stack first, and restores them before returning.

Obviously, the called procedure also takes responsibility to:
- allocate any needed space on the stack for local data
- place the return value onto the stack

In some situations, it is useful for the caller to also maintain the value that $sp held when the call was made, called the frame pointer. The register $fp would be used for this purpose.

Procedure Example 1

Let's implement a MIPS procedure to get a single integer input value from the user and return it:

```
get_integer:
    # Prompt the user to enter an integer value. Read and return it. It takes no parameters.
    li $v0, 4       # system call code for printing a string = 4
    la $a0, prompt # address of string is argument 0 to print_string
    syscall        # call operating system to perform print operation
    li $v0, 5       # get ready to read in integers
    syscall        # system waits for input, puts the value in $v0
    jr $ra
```

Since this doesn't use any registers that it needs to save, there's no involvement with the run-time stack.
Here's a sample header comment for this procedure:

```
###################################################
# get_integer
# Prompts user for an integer, reads it, returns it.
#
# Arg registers used: none
# Temp registers used: none
#
# Pre: a global string, labeled "prompt" exists
# Post: $v0 contains the value entered by the user
# Returns: value entered by the user
#
# Called by: main
# Calls: none
```

Since this procedure is extremely simple, the header is rather dull.

---

Since the procedure does not take any parameters, the call is simple. The return value will, by convention, have been placed in $v0.

```
... .data  # Data declaration section
    prompt: .asciiz "Enter an integer value\n"

.text

main:    # Start of code section
    jal    get_integer  # Call procedure
    move   $s0, $v0     # Put returned value in "save" reg

...  
```
Procedure Example 2

Let's design a procedure to take some integer parameters and compute a value from them and return that value. Say the expression to be computed is:

\[(a + b) - (c + d)\]

Then the caller needs to pass four arguments to the procedure; the default argument registers are sufficient for this.

The procedure only returns a single one-word value, so the default register is enough.

The procedure will use at least two registers to store temporary values while computing the expression, and a third register to hold the final result.

The procedure will use $t0$ and $t1$ for the temporaries, and $s0$ for the result.

(This could be done with fewer registers, but it's more illustrative this way.)

This one's a little more interesting:

```assembly
proc_example:
    addi $sp, $sp, -4    # adjust stack pointer to make room for 1 item
    sw $s0, 0($sp)      # save the value that was in $s0 when the call occurred
    add $t0, $a0, $a1   # $t0 = g + h
    add $t1, $a2, $a3   # $t1 = i + j
    sub $s0, $t0, $t1   # $s0 = (g + h) - (i + j)
    move $v0, $40        # put return value into $v0
    lw $s0, 0($sp)      # restore value of $s0
    addi $sp, $sp, 4     # restore the stack pointer
    jr $ra # jump back to the return address
```
### Procedure Call Example 2

This time we must place the parameters (presumably obtained by calling the procedure `get_integer`), into the default argument registers.

```assembly
  ...  move $a0, $s0      # position the parameters
  move $a1, $s1
  move $a2, $s2
  move $a3, $s3
  jal proc_example      # make the call
  move $a0, $v0         # return value will be in $v0
  li $v0, 1             # system call code for print_int
  syscall               # print it
  ...  
```

### Procedure Documentation

Here's another sample header comment:

```plaintext
# get_integer
# Computes and returns (a + b) - (c + d).
# Arg registers used: $a0 - $a3
# Tmp registers used: $t0, $t1
# Pre:       $a0 = a, $a1 = b, $a2 = c, $a3 = d
# Post:      $v0 contains (a + b) - (c + d)
# Returns:   value ($a0 + $a1) - ($a2 + $a3)
# Called by: main
# Calls:     none
```
Consider the following C implementation of bubblesort:

```c
void BubbleSort(int* L, int Sz) {
    int Stop, // upper limit for pass
        Curr, // index of current element
        Next, // index of next element
        Temp; // temp space for swapping
    for (Stop = Sz - 1; Stop > 0; Stop--) {
        for (Curr = 0; Curr < Stop; Curr++) {
            Next = Curr + 1;
            if (L[Curr] > L[Next]) {
                Temp = L[Curr];
                L[Curr] = L[Next];
                L[Next] = Temp;
            }
        }
    }
}
```

We need to map arguments and variables to registers, and identify any additional registers needed:

```c
void BubbleSort(int* L, int Sz) {
    int Stop, // upper limit for pass
        Curr, // index of current element
        Next, // index of next element
        Temp; // temp space for swapping
    for (Stop = Sz - 1; Stop > 0; Stop--) {
        for (Curr = 0; Curr < Stop; Curr++) {
            Next = Curr + 1;
            if (L[Curr] > L[Next]) {
                Temp = L[Curr];
                L[Curr] = L[Next];
                L[Next] = Temp;
            }
        }
    }
}
```

- $a0$ and $a1$, respectively
- $t3$: upper limit for pass
- $t0$: counter for inner loop
- $t2$: address of current elem
- no need for these
- $t7$: current value
- $t8$: next value
Interface and Outer Loop

The previous deliberations lead to:

```
       # initialize outer loop limit
  addi $t3, $a1, -1

outer:
    # outer bubble-sort loop
    bge $zero, $t3, outer_end
    li $t0, 0                # initialize inner loop counter
    move $t2, $a0              # set address of first elem

    # . . . inner loop goes here
    addi $t3, $t3, -1          # decrement outer loop limit
    j outer

outer_end:
    jr $ra # return to caller
```

Inner Loop

The previous deliberations lead to:

```
    # inner bubble-sort loop
    bge $t0, $t3, inner_end # reached stop index yet?
    lw $t7,  ($t2)              # get value at current index
    lw $t8, 4($t2)              # get value at "next" index
    ble $t7, $t8, no_swap # see if swap is needed
    sw $t8,  ($t2)
    sw $t7, 4($t2)

no_swap:
    addi $t0, $t0, 1           # increment inner loop counter
    addi $t2, $t2, 4           # and pointers
    j inner # restart inner loop

inner_end:
    . . .
```
Of course, we know that the solutions of the equation \( ax^2 + bx + c = 0 \)
can be found by using the formula
\[
-x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]
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
## Selected MIPS Floating-Point Instructions

MIPS assembly includes an impressive collection of instructions for performing computations with floating-point values; the latest release of SPIM supports most of them.

- `.float` directives for declaring 32- and 64-bit floating-point data
- `.double`
- `lwc1` load single from memory to FP register (aka `l.s`)
- `ldc1` load double from memory to FP register (note the pattern)
- `swc1` store single from FP register to memory (aka `s.s`)
- `sdc1` store double from FP register to memory
- `add.[s|d]` add single|double values
- `mul.[s|d]` multiple single|double values
- `sub.[s|d]` subtract single|double values
- `div.[s|d]` divide single|double values
- `neg.[s|d]` negate single|double value
- `sqrt.[s|d]` compute the non-negative square root of the argument

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`
- `bcl[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.
Quadratic Solver Design

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:

```
  a
  b
  c
  ...
```

Quadatic 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

Procedures 25

```
# initial code
quad_solver:
.data
 two:    .word 2
 four:   .word 4
 .text
   l.s $f0, 8($sp)   # retrieve coefficients 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?
   bc1f isOK                     # if not, compute solutions
   li $v0, 1                     # else, set error code
   jr $ra                       # and quit

# continuation
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       # f10 = one root
   neg.s $f9, $f9
   add.s $f11, $f7, $f9
   div.s $f11, $f11, $f8       # f11 = other root
   jr $ra
```

Quadratic Solver

Procedures 26

```
# 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       # f10 = one root
   neg.s $f9, $f9
   add.s $f11, $f7, $f9
   div.s $f11, $f11, $f8       # f11 = other root
   jr $ra
```

Procedures

get_coefficient

```asm
# 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
```

---

Leaf vs Non-leaf Procedures

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:

```asm
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

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