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:

```
jal    <address>  # $ra = PC + 4
                  # 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 8 bytes later…

Returning from the procedure requires transferring execution to the return address the `jal` instruction placed in `$ra`:

```
jr    $ra          # 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…

Procedure Example 1


```
#################################################################
# Returns largest value in an array of integers.
# Pre:   $a0 points to the first array element
#        $a1 is the number of elements in the array
# Post:  $a0 points one past the end of the array
#        $a1 is unchanged
#        $v0 is the largest value in the array
# Uses:  $t0, $t1
# Calls: none
FindMax:                              # label to jump to (proc name)
      li    $t0, 4                    # word size
      mul   $t0, $a1, $t0             # calculate offset to end of array
      add   $t0, $a0, $t0             # calculate stop address
      lw    $v0, 0($a0)               # initial max value is 0-th element
      addi  $a0, $a0, 4               # step pointer to next element

fmaxLoop:
      bge   $a0, $t0, fmaxDone        # see if we're past the array end
      lw    $t1, 0($a0)               # get next array element
      bge   $v0, $t1, noNewMax        # no new max
      move  $v0, $t1                  # reset max
noNewMax:
      addi  $a0, $a0, 4               # step pointer to next element
      j     fmaxLoop                  # return to loop test
fmaxDone:
      jr    $ra                       # return to caller
```

Procedure Call Example 1

We need to prepare the parameters, and use `jal` to make the call:

```
. . .
 .data
Size: .word 10
List: .word 78, 23, 41, 55, 18, 37, 81, 49, 74, 89

      .text
main:
      la    $a0, List               # $a0 is a pointer to the array
      lw    $a1, Size               # $a1 is the array size

      jal   FindMax                 # call FindMax procedure
. . .
```

Procedure Documentation

The header comment is not very different from what you'd use in any language:

```
##############################################################################
# Returns largest value in an array of integers.
#
# Pre:   $a0 points to the first array element
#        $a1 is the number of elements in the array
# Post:  $a0 points one past the end of the array
#
#        $a1 is unchanged
#        $v0 is the largest value in the array
#
# Uses:  $t0, $t1
# Calls: none
#
```

It **is** important to list all registers that are modified by the procedure.

Procedure-related 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, if necessary
- ensure the value of the stack pointer is the same after the call as it was before the call

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.