Laboratory Exercise 9

Procedure calls, stack and parameters

Goals

After this laboratory exercise, students will know how to invoke a procedure and get the returned values of the procedure. In addition, students need to understand the mechanism of stack and how to pass procedure parameters to stack.

Preparation

Students should review the textbook Computer Organisation and Design by Patterson & Henessy (Section 6.1).

Procedure calls

A procedure is a subprogram that, when invoked, is used to perform a specific task. When a procedure finishes, it returns to the point of origin with the desired results.

To call a procedure, we use the *jal* instruction. In particular, this instruction performs the control transferring (unconditional jump) to the starting address of the procedure, and also saves the return address in register *$ra*. Note that in MIPS assembly, a procedure is associated with a symbolic name that denotes its starting address.

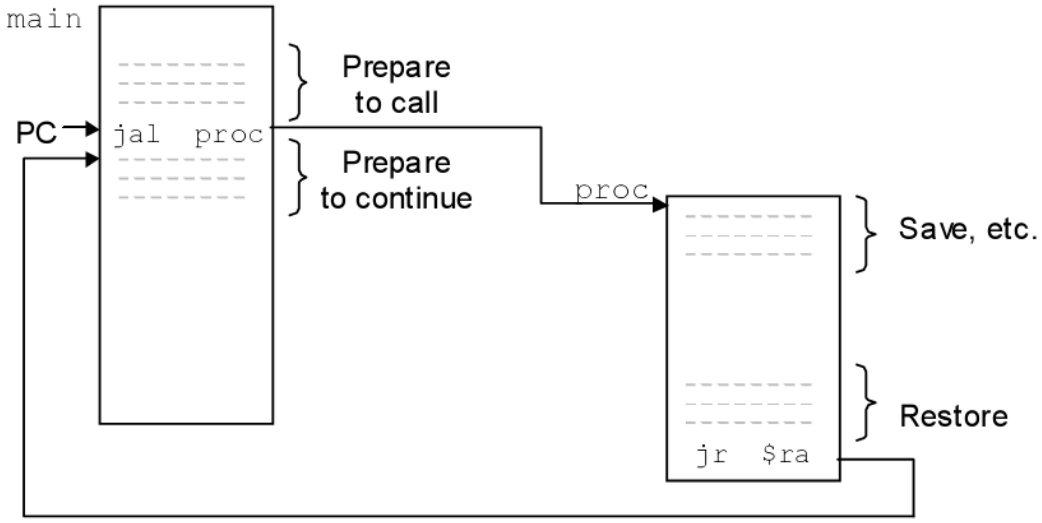

Figure 5. Relationship between the main program and a procedure

Sample Code and Assignments

Sample Code 1

The following program is used to find the absolute value of an integer. This program uses register *$a0* for the input argument and register *$v0* for the returned result. Read this example carefully.

```
#Laboratory Exercise 9 Sample Code 1
.text
main:   li      $a0,-45         #load input parameter
        jal     abs             #jump and link to abs procedure
        nop
        add     $s0, $zero, $v0
        li      $v0,10          #terminate
        syscall
endmain:
#---------------------------------------------------------------
-----
# function abs
# param[in]    $a1     the interger need to be gained the
absolute value
# return       $v0     absolute value
#---------------------------------------------------------------
-----
abs:
      sub   $v0,$zero,$a1     #put -(a0) in v0; in case (a0)<0

      bltz  $a1,done          #if (a0)<0 then done
      nop
      add   $v0,$a1,$zero     #else put (a0) in v0
done:
      jr    $ra
```

Sample Code 2

The following program is used to find the largest element among three integers. The input intergers are passed to the *max* procedure through registers *$a0*, *$a1*, and *$a2*. Read this example carefully and try to explain the sample code line by line.

```
#Laboratory Exercise 9, Sample Code 2
.text
main:  li    $a0,2            #load test input
       li    $a1,6
       li    $a2,9
       jal   max              #call max procedure
       nop
endmain:
#---------------------------------------------------------------
-
#Procedure max: find the largest of three integers
#param[in]  $a0  integers
#param[in]  $a1  integers
#param[in]  $a2  integers
#return     $v0   the largest value
#---------------------------------------------------------------
-
max:  add     $v0,$a0,$zero   #copy (a0) in v0; largest so far
      sub     $t0,$a1,$v0     #compute (a1)-(v0)
      bltz    $t0,okay        #if (a1)-(v0)<0 then no change
      nop
      add     $v0,$a1,$zero   #else (a1) is largest thus far
okay: sub     $t0,$a2,$v0     #compute (a2)-(v0)
      bltz    $t0,done        #if (a2)-(v0)<0 then no change
      nop
      add     $v0,$a2,$zero   #else (a2) is largest overall
done: jr      $ra             #return to calling program
```

Sample Code 3

The following program demonstrates the push and pop operations in the stack mechanism. In particular, the value in two registers *$s0* and *$s1* will be swapped using stack. Read this example and pay attention to the stack pointer adjustment and the order of the push and pop operations (i.e. *sw* and *lw* instructions).

```
#Laboratory Exercise 9, Sample Code 3
.text
push: addi    $sp,$sp,-8       #adjust the stack pointer
      sw      $s0,4($sp)       #push $s0 to stack
      sw      $s1,0($sp)       #push $s1 to stack
work: nop
      nop
      nop
pop:  lw      $s0,0($sp)       #pop from stack to $s0
      lw      $s1,4($sp)       #pop from stack to $s1
      addi    $sp,$sp,8        #adjust the stack pointer
```

Parameters and results

Let's think of the following questions:

1. How to pass more than four parameters to a procedure or how to receive more than two results from a procedure?
2. Where does a procedure save its parameters and/or immediate results when it calls another procedure (i.e. a nested call)?

One solution to the aforementioned problems is to use the stack mechanism. In particular, before a procedure call, the caller pushes the contents of any register referenced by the procedure onto the top of the stack; any additional procedure arguments are saved in the following positions in the stack. When the procedure terminates, the caller will restore the saved registers to continue its computation. Therefore, the procedure must save the content of the stack pointer *($sp*) before any modification of the stack pointer, and restore it at the end of execution. This is done by saving the stack pointer into the frame pointer (*$fp*). However, the old content of the frame pointer also must be saved before. Therefore, *$fp* and *$sp* together "frame" the stack area used by the current procedure.

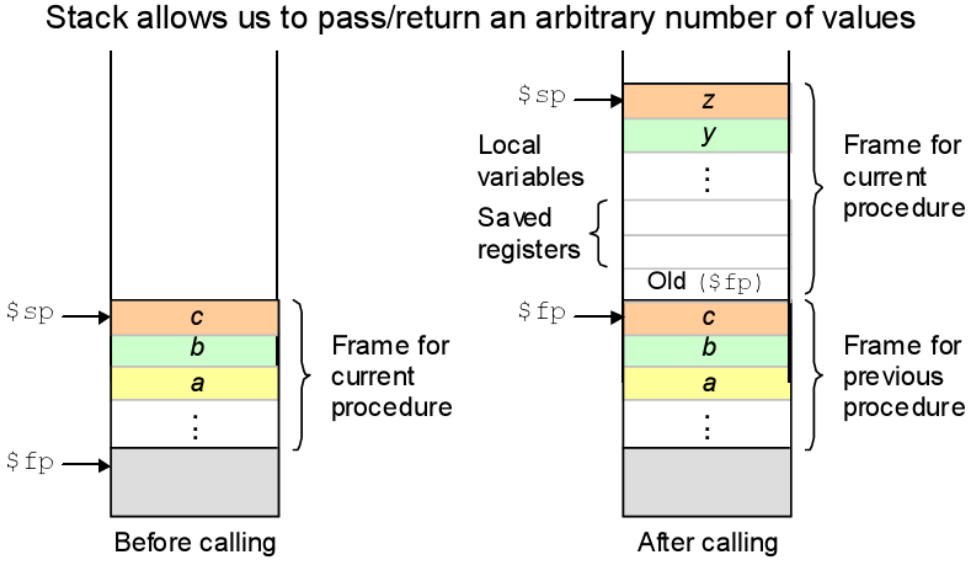

Figure 6. Use of the stack by a procedure

Sample Code 4

The following program is a recursive procedure for computing *n!*. Read this example carefully and pay attention to registers *$sp* and *$fp* at the start/end of each recursive step.

```
#Laboratory Exercise 9, Sample Code 4
.data
Message: .asciiz  "Ket qua tinh giai thua la: "

.text
main:  jal    WARP

print: add    $a1, $v0, $zero # $a0 = result from N!
       li     $v0, 56
       la     $a0, Message
       syscall
quit:  li     $v0, 10         #terminate
       syscall
endmain:

#---------------------------------------------------------------
#Procedure WARP: assign value and call FACT
#---------------------------------------------------------------
WARP:   sw     $fp,-4($sp)   #save frame pointer (1)
        addi   $fp,$sp,0     #new frame pointer point to the top (2)
        addi   $sp,$sp,-8    #adjust stack pointer (3)
        sw     $ra,0($sp)    #save return address (4)

        li     $a0,6         #load test input N
        jal    FACT          #call fact procedure
        nop

        lw     $ra,0($sp)    #restore return address (5)
        addi   $sp,$fp,0     #return stack pointer  (6)
        lw     $fp,-4($sp)   #return frame pointer  (7)
        jr     $ra
wrap_end:

#---------------------------------------------------------------
#Procedure FACT: compute N!
#param[in]  $a0  integer N
#return     $v0   the largest value
#---------------------------------------------------------------
FACT:  sw     $fp,-4($sp)        #save frame pointer
       addi   $fp,$sp,0          #new frame pointer point to stack's
top
       addi   $sp,$sp,-12        #allocate space for $fp,$ra,$a0 in
stack
       sw     $ra,4($sp)         #save return address
       sw     $a0,0($sp)         #save $a0 register

       slti   $t0,$a0,2          #if input argument N < 2
       beq    $t0,$zero,recursive#if it is false ((a0 = N) >=2)
       nop
       li     $v0,1              #return the result N!=1
       j      done
       nop
recursive:
       addi   $a0,$a0,-1         #adjust input argument
       jal    FACT               #recursive call
       nop
       lw     $v1,0($sp)         #load a0
```

```
        mult    $v1,$v0             #compute the result
        mflo    $v0
done:   lw      $ra,4($sp)          #restore return address
        lw      $a0,0($sp)          #restore a0
        addi    $sp,$fp,0           #restore stack pointer
        lw      $fp,-4($sp)         #restore frame pointer
        jr      $ra                 #jump to calling
fact_end:
```

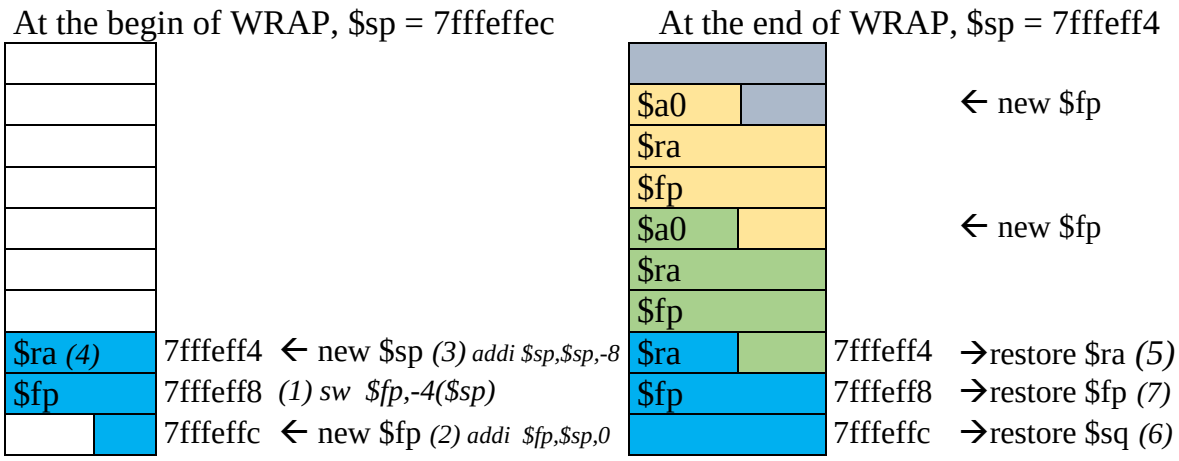

Figure 7. Invoking procedure process

Assignment 1

Create a new project in the MARs simulator to implement the program in Sample Code 1. Change the input parameters and observe the memory state by running the program step by step. Pay attention to registers *$pc* and *$ra* to clarify the invoking procedure process (Refer to Figure 7).

Assignment 2

Create a new project in the MARs simulator to implement the program in Sample Code 2. Change input parameters (in registers *$a0*, *$a1*, *$a2*), and observe the memory state running the program step by step. Pay attention to register *$pc* and *$ra* to clarify the invoking procedure process (Refer to Figure 7).

Assignment 3

Create a new project in the MARs simulator to implement the program in Sample Code 3. Pass the test inputs through registers *$s0* and *$s1*, observe the run process and the stack pointer. Go to the memory space pointed by register *$sp* to see the push and pop operations in detail.

Assignment 4

Create a new project in the MARs simulator to implement the program in Sample Code 4. Pass the test input through register *$a0*, and the test result in register *$v0*. Run this program step by step and observe the change in registers *$pc*, *$ra*, *$sp* and *$fp*. Given that n=3, draw the state of the stack when calling this recursive program.

Assignment 5

Write a procedure to find the largest element and the smallest element (and their positions as well) in a list of 8 elements. Assume that the 8 elements are stored in regsiters *$s0 ~ $s7*.

For example:
Largest: 9,3 → The largest element is stored in *$s3*, largest value is 9

Smallest: -3,6 → The smallest element is stored in *$s6*, smallest value is -3
Hints: using stack to pass the arguments and return results.

Questions

- What registers does the Caller need to save by convention?
- What registers does the Callee need to save by convention?
- In the *push* label of Sample Code 3, could we change the order of stack pointer adjustment operation and the *sw* operation? If yes, what should we have to modify?
- What is stack pointer?
- What is frame pointer?