MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

If you did want to speed up a high-level program by using assembly language, you would compile the program with a profiling option, and then run the program with real data. Having profiling turned on causes the program to track where it spends its execution time, and generates a report of the program's profile. Usually, over 85% of a program's time is spent executing a single subprogram. Thus, you can write just this subprogram in assembly language and leave the rest of the program in the high-level language (HLL). To correctly have the HLL program call your assembly-language subprogram, your assembly-language subprogram must follow the run-time stack and *register conventions* established for the processor. The register conventions are the rules about how the registers should be used.

Compiler uses registers to avoid accessing the run-time stack in memory as much as possible. Registers can be used for local variables, parameters, the return address, and the function-return value. Unfortunately, the number of registers is limited. When a subprogram is called, some of the register values might need to be saved ("spilled") on the stack to free up some registers for the subprogram to use.

Different machines use one of several standard conventions for spilling registers:
1) caller save - before the call, caller saves the register values it needs after execution returns from the subprogram
2) callee save - subprogram saves and restores any register it uses in its code
3) some combination of caller and callee saved (USED BY MIPS)

The following table shows the MIPS register conventions. Each register can be referenced to by its number or its convention name, e.g., $4 as $a0 for an "argument"/parameter register. The caller of the subprogram would place the parameter value in $a0 and call the subprogram. The subprogram has the parameter value since it is in the register. Thus, avoiding pushing it on the run-time stack in the slow memory.

MIPS Register Conventions			
Reg. #	**Convention Name**	**Role in Procedure Calls**	**Comments**
$0	$zero	constant value zero	Cannot be changed
$1	$at	Used by assembler to implement psuedoinstructions	DON'T USE
$2, $3	$v0, $v1	Results of a function	
$4 - $7	$a0 - $a3	First 4 arguments to a procedure	
$8 - $15, $24, $25	$t0 - $t9	Temporary registers (not preserved across call)	Caller-saved registers - subprogram can use them as scratch registers, but it must also save any needed values before calling another subprogram.
$16 - $23	$s0 - $s7	Saved temporary (preserved across call)	Callee-saved registers - it can rely on an subprogram it calls not to change them (so a subprogram wishing to use these registers must save them on entry and restore them before it exits)
$26, $27	$k0, $k1	Reserved for the Operating System Kernel	DON'T USE
$28	$gp	Pointer to global area	
$29	$sp	Stack pointer	Points to first free memory location above stack
$30	$fp/$s8	Frame pointer (if needed) or another saved register	$fp not used so use as $s8
$31	$ra	Return address (used by a procedure call)	Receives return addr. on *jal* call to procedure

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

The general steps for using the MIPS register conventions are listed below.

Using MIPS Calling Convention	
Caller Code **(caller of the subprogram)**	**Callee Code** **(the subprogram itself)**
. . . 1) save on stack any $t0 - $t9 and $a0 - $a3 that are needed upon return 2) place arguments to be passed in $a0 - $a3 with additional parameters pushed onto the stack 3) jal ProcName # saves return address in $ra 4) restore any saved registers $t0 - $t9 and $a0 - $a3 from stack	. . . 1) allocate memory for frame by subtracting frame size from $sp 2) save callee-saved registers ($s0 - $s7) if more registers than $t0 - $t9 and $a0 - $a3 are needed 3) save $ra if another procedure is to be called **. . . code for the callee** 4) for functions, place result to be returned in $v0 - $v1 5) restore any callee-saved registers ($s0 - $s7) from step (2) above 6) restore $ra if it was saved on the stack in step (3) 7) pop stack frame by adding frame size to $sp 8) return to caller by "jr $ra" instruction

Lets reconsider the CalculatePowers program and examine how to apply the MIPS register conventions:

High-level Language Programmer's View of CalculatePowers

main:	**CalculatePowers**(integer numLimit, integer powerLimit)	integer function **Power**(integer n, integer e)
maxNum = 3 maxPower = 4	integer num, pow	integer result if e = 0 then

CalculatePowers(maxNum, maxPower)
(*)
. . .
end main
	for num := 1 to numLimit do
	for pow := 1 to powerLimit do
	
	print num " raised to " pow " power is "
	Power(num, pow)
	end for pow
	end for num
		result = 1
		else if e = 1 then
		result = n
		else
		result = Power(n, e - 1)* n
		end if
		return result
		end Power

I would recommend asking your self the following questions to determine which registers to use when applying the register conventions.

1) Using the MIPS register conventions, what registers would be used to pass the parameters (maxNum and maxPower) to CalculatePowers?

The first parameter is always passed in $a0, the second parameter in $a1, etc. If there are more than four parameters, then additional parameters

are pushed onto the run-time stack. Thus, we'll use the following register allocation:

maxNum	maxPower
$a0	$a1

The main program code that calls CalculatePowers would be:

```
main:
          ...
          lw   $a0, maxNum          # $a0 contains maxNum
          lw   $a1, maxPower        # $a1 contains maxPower
          jal  CalculatePowers
```

The jump-and-link (`jal`) instruction acts an unconditional jump instruction, but it also saves the address of the instruction after the jal to register $ra so execution can return there when CalculatePowers returns. Having the return-address saved to a register avoids the need to save it to the run-time stack.

When CalculatePowers starts execution its formal parameters, numLimit and powerLimit, will be in registers $a0 and $a1, respectively. Since CalculatePowers calls the Power function which takes two parameters, both register $a0 and $a1 must eventually be used for this purpose. You want to decide if either numLimit or powerLimit is needed across the call to Power. If so, we must save their value(s) before calling Power. One way to save their value is to move them into an s-register which is maintained across the call to a subprogram (i.e., the subprogram will not change the s-registers if it is following the register conventions). In writing the code for a subprogram, the second question I ask myself is:

2) Using the MIPS register conventions, which of these parameters ("numLimit", "powerLimit", or both of them) should be moved into s-registers?
(**NOTE: Use an s-register for any value you still need after you come back from a subprogram/function/procedure call, e.g., call to "Power"**)

The call to Power is part of the inner-for-loop body with numLimit and powerLimit both being needed after the call (so they can be compared to the loop control variables). Thus, both should be saved to s-registers: $a0 can be moved to $s0 and $a1 can be moved to $s1.

For the local variables, num and pow, we need to ask ourself a similar question:
3) Using the MIPS register conventions, what registers should be used for each of the local variables?

Since both variables are used as loop-control variables with Power being called as part of the inner-for-loop body, both variable must maintain their value across the call to Power. Thus, s-registers should be used for both. Since $s0 and $s1 are already being used for numLimit and powerLimit, we can use $s2 and $s3 as:

num	pow
$s2	$s3

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

Before we can use $s0 to $s3 in CalculatePowers, we need to save their values to the run-time stack. After all, "main" might be storing something in these s-registers, and by convention CalculatePowers should not be allowed to change "main's" values in s-registers. The code that starts the CalculatePowers subprogram would be:

```
CalculatePowers:    # parameters:    $a0 contains numLimit, and $a1 contains powerLimit

          sub  $sp, $sp, -20       # move stack pointer, $sp, "up" to make room for the call-frame
          sw   $ra, 4($sp)         # push return address onto stack
          sw   $s0, 8($sp)         # push the caller's s-register values onto the stack
          sw   $s1, 12($sp)
          sw   $s2, 16($sp)
          sw   $s3, 20($sp)
                              # save a-registers to s-registers so they don't get wiped out
          move $s0, $a0       # save numLimit in $s0
          move $s1, $a1       # save powerLimit in $s1
...
```

To create room on the run-time stack for CalculatePowers' call-frame, we subtract 20 bytes from the stack-pointer, $sp register, which is enough for 5 registers. The registers saved are the s-registers $s0 to $s3 and the $ra register which contains the return address back in the "main." We save the $ra on the stack since the "`jal Power`" instructions in the subprogram body would wipe it out.

Since CalculatePowers calls Power, we start over with the same set of questions for the call to Power and the code for Power.

1) Using the MIPS register conventions, what registers would be used to pass each of the following parameters to Power:

num	pow
$a0	$a1

The CalculatePowers' code that calls Power would be:

```
CalculatePowers:
          ...
          move $a0, $s2       # call Power(num, pow), where num is in $s2 and pow is in $s3
          move $a1, $s3
          jal  Power
          ...
```

Writing the Power function is a little trick. Since it is recursive and it only calls itself, we can "bend the register conventions" alittle bit to improve efficiency.

2) Using the MIPS register conventions, which of these parameters ("n", "e", or both of them) should be moved into s-registers?

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

Normally, I'd look at the recursive call "result = Power(n, e - 1)* n", and think that parameter n is needed after we return from Power so we can multiple it. Thus, I'd want to move the original parameter n in $a0 to an s-register, so the call to Power does not wipe it out. However, the first parameter in the recursive call is the value of n, so we can just leave $a0 as the value of n throughout subprogram Power. If this is confusing, you might look at the HLL run-time stack diagram on page 6 of the "MIPS Supplement". Notice that n's value is unchanged in each call-frame.

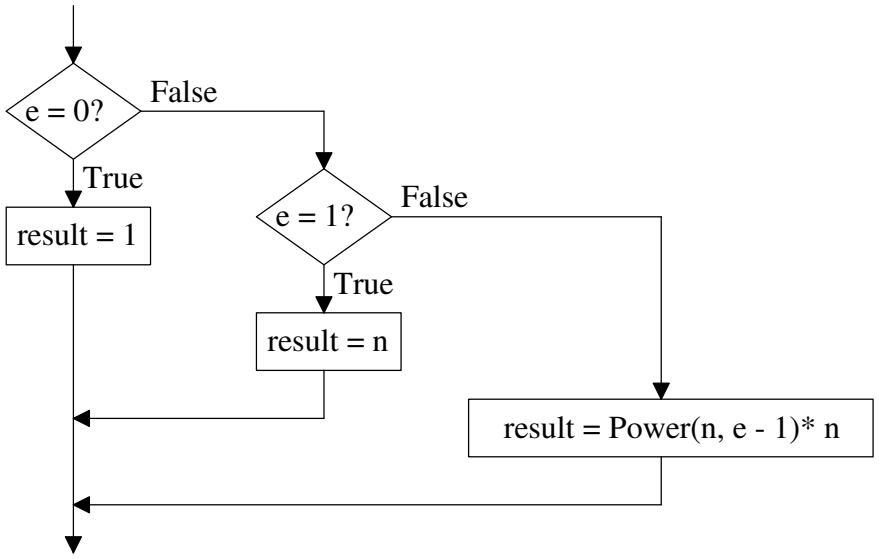
The parameter e is initially in $a1. Since e's value is not needed after the recursive call to Power, it does not need to be saved to an s-register.

3) Using the MIPS register conventions, what register should be used for the local variable "result"?

Since "result" has no value before the recursive call to Power, we don't need to use a s-register. The value of "result" is returned as the function value so using $v0 makes the most sense. The flow-chart of the Power function clearly shows that "result" does not have a value before the recursive call. Suppose e = 1, then the "result = n" assignment statement would be executed, but the recursive call would not be performed.

HLL Power code:

```
integer function Power( integer n, integer e)
integer result
    if e = 0 then
        result = 1
    else if e = 1 then
        result = n
    else
        result = Power(n, e - 1)* n
    end if
    return result
end Power
```



The Power function in MIPS assembly language using the above decisions is given below. Since neither parameter nor the local variable "result" needs to be saved to an s-register, the only thing to save on the run-time stack is the return-address register, $ra.

```
Power:                              # $a0 contains n (we never change it during the
                                    #        recursive calls so we don't need to save it)
                                    # $a1 contains e
          sub  $sp, $sp, -4         # make room for the call-frame
          sw   $ra, 4($sp)          # save $ra on stack

if:
          bne  $a1, $zero, else_if
          li   $v0, 1               # $v0 contains result
          j    end_if
else_if:
          bne  $a1, 1, else
          move $v0, $a0
          j    end_if
else:                               # first parameter is still n in $a0
          addi $a1, $a1, -1         # put second parameter, e-1, in $a1
          jal  Power                # returns with value of Power(n, e-1) in $v0
          mul  $v0, $v0, $a0        # result = Power(n, e-1) * n
end_if:
          lw   $ra, 4($sp)          # restore return addr. to $ra
          addi $sp, $sp, 4          # pop call frame from stack
          jr   $ra
end_Power:
```

Notice at the end of Power, we must

- restore the saved register(s) (here only the $ra register),
- restore the stack pointer to its original position before the call, and
- jump-register (`jr $ra`) back to the return address in register $ra.

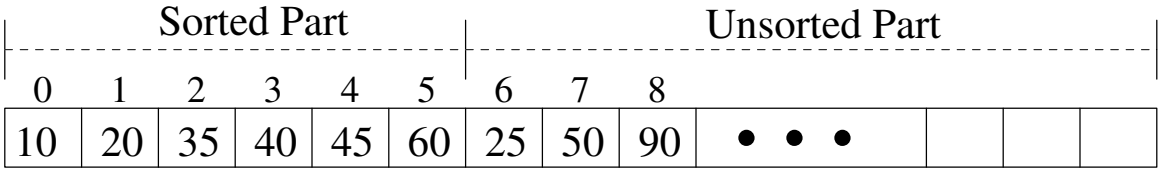
Complete code for the above example, including printing of the output, is included at the end of this Supplement. Before we look at this code, lets practice using the register conventions on the following Insertion sort example.

Insertion sort is a *simple sort*. All simple sorts consist of two nested loops where:

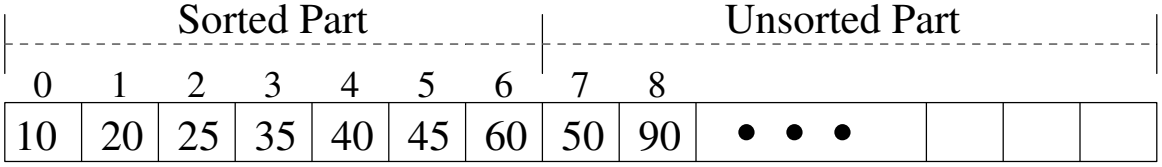
- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
- the inner loop's job is to do the work to extend the sorted part's size by one.

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

After several iterations of the outer loop, an array might look like:



Insertion sort takes the "first unsorted element" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the array would look like:



The below code, splits off the inner-loop into its own Insert subprogram. The array is passed by sending its starting/base address as a parameter.

```
main:

integer scores [100];
integer n;  // # of elements

InsertionSort(scores, n)

(*)
. . .

end main
```

```
InsertionSort(numbers - address to integer array,
              length - integer)

    integer firstUnsortedIndex
    for firstUnsortedIndex = 1 to (length-1) do
        Insert(numbers, numbers[firstUnsortedIndex],
               firstUnsortedIndex-1);
    end for

end InsertionSort
```

```
Insert(numbers - address to integer array,
       elementToInsert - integer,
       lastSortedIndex - integer) {
    integer testIndex;
    testIndex = lastSortedIndex;
    while (testIndex >=0) AND
          (numbers[testIndex] > elementToInsert ) do
        numbers[ testIndex+1 ] = numbers[ testIndex ];

        testIndex =  testIndex - 1;
    end while
    numbers[ testIndex + 1 ] = elementToInsert;
end Insert
```

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

Try to apply the MIPS register conventions by answering the "standard" questions. On the next page, I'll supply my answers to these questions.

1) Using the MIPS register conventions, what registers would be used to pass each of the following parameters from main to InsertionSort:

scores	n

2) Using the MIPS register conventions, which of these parameters ("numbers", "length", or both of them) should be moved into s-registers?

3) Using the MIPS register conventions, what registers should be used for the local variable "firstUnsortedIndex"?

1) Using the MIPS register conventions, what registers would be used to pass each of the following parameter values to Insert:

numbers	numbers[firstUnsortedIndex]	firstUnsortedIndex-1

2) Using the MIPS register conventions, which of these parameters ("numbers", "elementToInsert", or "lastSortedIndex") should be moved into s-registers?

3) Using the MIPS register conventions, what registers should be used for the local variable "testIndex"?

Try to write the code for main, InsertionSort, and Insert in MIPS assembly language.

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

My answers to these questions:

1) Using the MIPS register conventions, what registers would be used to pass each of the following parameters from main to InsertionSort:

scores	n
$a0	$a1

Because the first parameter always is passed in $a0, and the second parameter is always passed in $a1. The array scores is passed by sending the base address in $a0.

2) Using the MIPS register conventions, which of these parameters ("numbers", "length", or both of them) should be moved into s-registers?

Both numbers and (length-1) need to be stored in s-registers since their values are needed each iteration of the for-loop, i.e., their values are needed across the call to Insert. We'll use $s0 for "numbers" and $s1 for the value of (length-1).

3) Using the MIPS register conventions, what registers should be used for the local variable "firstUnsortedIndex"?

The local variable firstUnsortedIndex needs to be stored in an s-register since its value is needed each iteration of the for-loop, i.e., its value is needed across the call to Insert. We'll use $s2 for "firstUnsortedIndex".

1) Using the MIPS register conventions, what registers would be used to pass each of the following parameter values to Insert:

numbers	numbers[firstUnsortedIndex]	firstUnsortedIndex-1
$a0	$a1	$a2

2) Using the MIPS register conventions, which of these parameters ("numbers", "elementToInsert", or "lastSortedIndex") should be moved into s-registers?

Since Insert does not call any subprograms, we can leave the parameters in the a-registers.

3) Using the MIPS register conventions, what registers should be used for the local variable "testIndex"?

Since Insert does not call any subprograms, we can use a "temporary" register, say $t0 for testIndex. Thus, Insert does not need to store anything on the run-time stack making it very efficient. The beauty of a combined caller-saved with callee-saved register convention.

MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

main:	**InsertionSort**(numbers - address to integer array, length - integer)	**Insert**(numbers - address to integer array, elementToInsert - integer, lastSortedIndex - integer) {

integer scores [100];
integer n; // # of elements
 	
InsertionSort(scores, n)
 	
(*)
. . .
 	
end main
	integer firstUnsortedIndex
	for firstUnsortedIndex = 1 to (length-1) do
	Insert(numbers, numbers[firstUnsortedIndex],
	firstUnsortedIndex-1);
	end for
	
	end InsertionSort
		integer testIndex;
		testIndex = lastSortedIndex;
		while (testIndex >=0) AND
		(numbers[testIndex] > elementToInsert) do
		numbers[testIndex+1] = numbers[testIndex];
		
		testIndex = testIndex - 1;
		end while
		numbers[testIndex + 1] = elementToInsert;
		end Insert

The code for main, InsertionSort, and Insert in MIPS assembly language is given below.

```
            .data
scores:     .word 20, 30, 10, 40, 50, 60, 30, 25, 10, 50
n:          .word 10

            .text
            .globl main
main:
            la $a0, scores
            lw $a1, n
            jal insertionSort

            li $v0, 10
            syscall
end_main:

################################################################################

insertionSort:
            sub $sp, $sp, 16        # setup call-frame
            sw $ra, 4($sp)          # save the return addr to get back to main
            sw $s0, 8($sp)          # save main's s-registers if it is
            sw $s1, 12($sp)         # using any
            sw $s2, 16($sp)

            move $s0, $a0           # save address of numbers in $s0
            sub $s1, $a1, 1         # save (length-1) in $s1
for_init:
            li $s2, 1               # save firstUnsortedIndex in $s2
for_loop:   bgt $s2, $s1, end_for
            move $a0, $s0           # fill actual parameters
```

```
            mul $t0, $s2, 4
            add $t0, $s0, $t0
            lw $a1, 0($t0)
            sub $a2, $s2, 1
            jal insert              # call insert subpgm
            addi $s2, $s2, 1
            j for_loop
end_for:
            lw $ra, 4($sp)          # restore return addr. and
            lw $s0, 8($sp)          # restore main's s-registers
            lw $s1, 12($sp)
            lw $s2, 16($sp)
            addi $sp, $sp, 16       # remove call-frame
            jr $ra                  # 'jump register' back to main
end_insertionSort:

#############################################################################################

insert:  # insert does not call any subpgms so we'll use $a and $t registers
         # nothing to save on the run-time stack
            move $t0, $a2           # use $t0 for testIndex
while:      blt $t0, 0, end_while
            mul $t1, $t0, 4
            add $t1, $a0, $t1
            lw $t2, 0($t1)
            ble $t2, $a1, end_while
            sw $t2, 4($t1)
            sub $t0, $t0, 1
            j while
end_while:
            mul $t1, $t0, 4
            add $t1, $a0, $t1
            sw $a1, 4($t1)
            jr $ra                  # 'jump register' back to insertionSort
end_insert:
```

PCSpim I/O Support

Access to Input/Output (I/O) devices within a computer system is generally restricted to prevent user programs from directly accessing them. This prevents a user program from accidentally or maliciously doing things like:

- reading someone else's data file from a disk
- writing to someone else's data file on a disk
- etc.

However, user programs need to perform I/O (e.g., read and write information to files, write to the console, read from the keyboard, etc.) if they are to be useful. Therefore, most computer systems require a user program to request I/O by asking the operating system to perform it on their behalf.

PCSpim uses the "syscall" (short for "system call") instruction to submit requests for I/O to the operating system. The register $v0 is used to indicate the type of I/O being requested with $a0, $a1, $f12 registers being used to pass additional parameters to the operating system. Integer results and addresses are returned in the $v0 register, and floating point results being returned in the $f0 register. The following table provides details of the PCSpim syscall usage.

Service Requested	**System call code passed in $v0**	**Registers used to pass additional arguments**	**Registers used to return results**
print_int	1	$a0 contains the integer value to print	
print_float	2	$f12 contains the 32-bit float to print	
print_double	3	$f12 (and $f13) contains the 64-bit double to print	
print_string	4	$a0 contains the address of the .asciiz string to print	
read_int	5		$v0 returns the integer value read
read_float	6		$f0 returns the 32-bit floating-point value read
read_double	7		$f0 and $f1 returns the 64-bit floating-point value read
read_string	8	$a0 contains the address of the buffer to store the string $a1 contains the maximum length of the buffer	
sbrk - request a memory block	9	$a0 contains the number of bytes in the requested block	$v0 returns the starting address of the block of memory
exit	10		

Below is the complete CalculatePowers program example using MIPS register conventions and PCSpim syscalls. Notice that strings use the ".asciiz" assembler directive to generate an array of ASCII characters that are "null" terminated (ASCII value 0).

```
# CalculatePowers subprogram example using MIPS register conventions and PCSpim syscalls

                .data
maxNum:         .word 3
maxPower:       .word 4
str1:           .asciiz " raised to "
str2:           .asciiz " power is "
str3:           .asciiz  "\n"                   # newline character

                .text
                .globl main
main:
                lw      $a0, maxNum             # $a0 contains maxNum
                lw      $a1, maxPower           # $a1 contains maxPower
                jal     CalculatePowers

                li      $v0, 10                 # system code for exit
                syscall


#################################################################################
CalculatePowers:                                # $a0 contains value of numLimit
                                                # $a1 contains value of powerLimit

                sub     $sp, $sp, -20           # save room for the return address
                sw      $ra, 4($sp)             # push return address onto stack
                sw      $s0, 8($sp)
                sw      $s1, 12($sp)
                sw      $s2, 16($sp)
                sw      $s3, 20($sp)

                move    $s0, $a0                # save numLimit in $s0
                move    $s1, $a1                # save powerLimit in $s1

for_1:
                li      $s2, 1                  # $s2 contains num
for_compare_1:
                bgt     $s2, $s0, end_for_1
for_body_1:

for_2:
                li      $s3, 1                  # $s3 contains pow
for_compare_2:
                bgt     $s3, $s1, end_for_2
for_body_2:
                move    $a0, $s2                # print num
                li      $v0, 1
                syscall
```

```
                la      $a0, str1               # print " raised to "
                li      $v0, 4
                syscall

                move    $a0, $s3                # print pow
                li      $v0, 1
                syscall

                la      $a0, str2               # print " power is "
                li      $v0, 4
                syscall

                move    $a0, $s2                # call Power(num, pow)
                move    $a1, $s3
                jal     Power

                move    $a0, $v0                # print result

                li      $v0, 1
                syscall

                la      $a0, str3               # print new-line character
                li      $v0, 4
                syscall

                addi    $s3, $s3, 1
                j       for_compare_2
end_for_2:

                addi    $s2, $s2, 1
                j       for_compare_1
end_for_1:

                lw      $ra, 4($sp)             # restore return addr. to $ra
                lw      $s0, 8($sp)             # restore saved $s registers
                lw      $s1, 12($sp)
                lw      $s2, 16($sp)
                lw      $s3, 20($sp)

                addi    $sp, $sp, 20            # pop call frame from stack
                jr      $ra
end_CalculatePowers:
```

```
#################################################################################
Power:                                  # $a0 contains n (we never change it during the
                                        #       recursive calls so we don't need to save it)
                                        # $a1 contains e
                sub     $sp, $sp, -4
                sw      $ra, 4($sp)     # save $ra on stack

if:
                bne     $a1, $zero, else_if
                li      $v0, 1          # $v0 contains result
                j       end_if
else_if:
                bne     $a1, 1, else
                move    $v0, $a0
                j       end_if
else:                                   # first parameter is still n in $a0
                addi    $a1, $a1, -1    # put second parameter, e-1, in $a1
                jal     Power           # returns with value of Power(n, e-1) in $v0
                mul     $v0, $v0, $a0   # result = Power(n, e-1) * n
end_if:
                lw      $ra, 4($sp)     # restore return addr. to $ra
                addi    $sp, $sp, 4     # pop call frame from stack
                jr      $ra
end_Power:
```

Snap-shot of the Console window after the program executes: