**Bahria University, Lahore Campus**

Department of Computer Sciences

Lab Journal 012

**(Spring 2023)**

|  |  |  |
| --- | --- | --- |
| Course: | **Computer Architecture & Organization Lab** |  |
| Course Code: | CEL 221 | Max Marks: 30 |
| Faculty’s Name: | Maryam Munawar | Lab Engineer: |

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Enroll No: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Objective(s):

Students will learn creating Stack and Function in MIPS

## Tool(s) used:

MIPS 4.5

MIPS Functions: A function (or a procedure) is a tool that programmers use to structure programs, to make them easier to understand, and to allow the function’s code to be reused. A function is a block of instructions that can be called and used when required at several different points in the program. The function that initiates the call to another function is known as the caller. The function that receives and executes the call is known as the callee. When the callee function finishes execution, control is transferred back to the caller function.

A function can receive parameters and return results. The parameters and results act as an interface between a function and the rest of the program.

To execute a function, the program must follow these steps:

1. The caller must put the parameters in a place where the callee function can access them
2. Transfer control to the callee function
3. Execute the callee function

4. The callee function must put the results in a place where the caller can access them

5. Return control to the caller (point of origin) next to where the call was made

Registers are the fastest place to pass parameters and return results. The MIPS architecture follows the following software conventions for passing parameters and returning results:

* $a0-$a3: four argument registers in which to pass parameters
* $v0-$v1: two value registers in which to return function results
* $ra: one return address register to return back to the caller

The jal (jump-and-link) instruction initiates the call to a function and the jr (jump register) instruction returns control back to the caller. To call a function, use the jal instruction as follows: jal label The jal instruction saves the return address in register $ra and jumps to the first instruction in the function after label. The return address is the address of the next instruction that appears after the jal instruction in the caller function. To return from a function, use the jr instruction as follows: jr $ra The jr instruction jumps to the address stored in $ra. It modifies the program counter PC register according to the value stored in register $ra. An example of a C function that checks whether a character ch is a lowercase letter or not is shown in Figure.

The function is translated into MIPS assembly language as shown to the right. The function islower assumes that the parameter ch is passed in register $a0. The function result is passed in register $v0

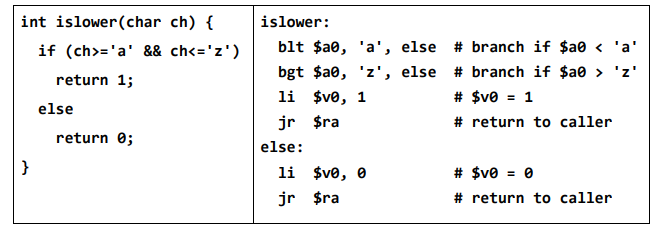


Figure 10

To call the function islower, the caller must first copy the character ch into register $a0 and then make the function call. This is shown in Figure

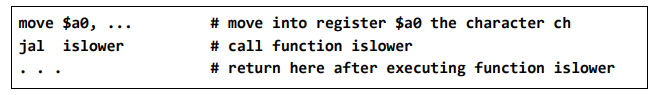


Figure 11

## The Stack Segment and the Stack Pointer Register:

Every program has three segments when it is loaded into memory by the operating system. There is the text segment where the machine language code is stored, the data segment where space is allocated for constants and variables, and the stack segment that provides an area that can be allocated and freed by functions. The programmer has no control over where these segments are located in memory. The stack segment can be used by functions for passing many parameters, for allocating space for local variables, and for saving and preserving registers across calls. Without the stack segment in memory, it would be impossible to write recursive functions, or pure functions that have no side effects. When a program is loaded into memory, the operating system initializes the stack pointer $sp (register $29) to the base address of the stack segment. The stack segment grows downwards towards lower memory addresses as shown in Figure

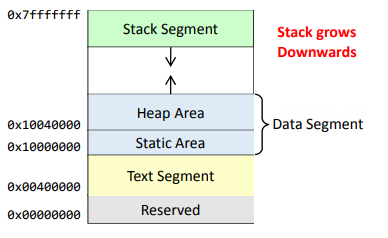


Figure 12

### Task 01. Time: 30 Minutes

### Subprogram PrintIntArray(array, size)

### {

### print("[")

### for (int i = 0; i < size; i++)

### {

### print("," + array[i])

### }

### print("]")

### }

### **Solution**

### .text

### .globl main

### main:

### la $a0, array\_base

### lw $a1, array\_size

### jal PrintIntArray

### jal Exit

### .data

### array\_size: .word 5

### array\_base:

### .word 12

### .word 7

### .word 3

### .word 5

### .word 11

### 

### .text

### # Subprogram: PrintIntArray

### # Purpose: print an array of ints

### # inputs: $a0 - the base address of the array

### # $a1 - the size of the array

### #

### PrintIntArray:

### addi $sp, $sp, -16 # Stack record

### sw $ra, 0($sp)

### sw $s0, 4($sp)

### sw $s1, 8($sp)

### sw $s2, 12($sp)

### 

### move $s0, $a0 # save the base of the array to $s0

### 

### # initialization for counter loop

### # $s1 is the ending index of the loop

### # $s2 is the loop counter

### move $s1, $a1

### move $s2, $zero

### la $a0 open\_bracket # print open bracket

### jal PrintString

### loop:

### # check ending condition

### sge $t0, $s2, $s1

### bnez $t0, end\_loop

### 

### sll $t0, $s2, 2 # Multiply the loop counter by

### # by 4 to get offset (each element

### # is 4 big).

### add $t0, $t0, $s0 # address of next array element

### lw $a1, 0($t0) # Next array element

### la $a0, comma

### jal PrintInt # print the integer from array

### 

### addi $s2, $s2, 1 #increment $s0

### b loop

### end\_loop:

### li $v0, 4 # print close bracket

### la $a0, close\_bracket

### syscall

### lw $ra, 0($sp)

### lw $s0, 4($sp)

### lw $s1, 8($sp)

### lw $s2, 12($sp) # restore stack and return

### addi $sp, $sp, 16

### jr $ra

### .data

### open\_bracket: .asciiz "["

### close\_bracket: .asciiz "]"

### comma: .asciiz ","

### .include "utils.asm"

### Task 02. Time: 30 Minutes

Pseudo code for this algorithm follws.

for (int i = 0; i < size-1; i++)

{

for (int j = 0; j < ((size-1)-i); j++)

{

if (data[j] > data[j+1])

{

swap(data, j, j+1)

}

}

}

swap(data, i, j)

int tmp = data[i];

data[i] = data[j];

data[j] = tmp;

}

**Solution**

The following assembly program implements the Bubble Sort matching the pseudo code algorithm in the previous section.

.text

.globl main

main:

la $a0, array\_base

lw $a1, array\_size

jal PrintIntArray

la $a0, array\_base

lw $a1, array\_size

jal BubbleSort

jal PrintNewLine

la $a0, array\_base

lw $a1, array\_size

jal PrintIntArray

jal Exit

.data

array\_size: .word 8

array\_base:

.word 55

.word 27

.word 13

.word 5

.word 44

.word 32

.word 17

.word 36

.text

# Subproram: Bubble Sort

# Purpose: Sort data using a Bubble Sort algorithm

# Input Params: $a0 - array

# $a1 - array size

# Register conventions:

# $s0 - array base

# $s1 - array size

# $s2 - outer loop counter

# $s3 - inner loop counter

BubbleSort:

addi $sp, $sp -20 # save stack information

sw $ra, 0($sp)

sw $s0, 4($sp) # need to keep and restore save registers

sw $s1, 8($sp)

sw $s2, 12($sp)

sw $s3, 16($sp)

move $s0, $a0

move $s1, $a1

addi $s2, $zero, 0 #outer loop counter

OuterLoop:

addi $t1, $s1, -1

slt $t0, $s2, $t1

beqz $t0, EndOuterLoop

addi $s3, $zero, 0 #inner loop counter

InnerLoop:

addi $t1, $s1, -1

sub $t1, $t1, $s2

slt $t0, $s3, $t1

beqz $t0, EndInnerLoop

sll $t4, $s3, 2 # load data[j]. Note offset is 4 bytes

add $t5, $s0, $t4

lw $t2, 0($t5)

addi $t6, $t5, 4 # load data[j+1]

lw $t3, 0($t6)

sgt $t0, $t2, $t3

beqz $t0, NotGreater

move $a0, $s0

move $a1, $s3

addi $t0, $s3, 1

move $a2, $t0

jal Swap # t5 is &data[j], t6 is &data[j=1]

NotGreater:

addi $s3, $s3, 1

b InnerLoop

EndInnerLoop:

addi $s2, $s2, 1

b OuterLoop

EndOuterLoop:

lw $ra, 0($sp) #restore stack information

lw $s0, 4($sp)

lw $s1, 8($sp)

lw $s2, 12($sp)

lw $s3, 16($sp)

addi $sp, $sp 20

jr $ra

# Subprogram: swap

# Purpose: to swap values in an array of integers

# Input parameters: $a0 - the array containing elements to swap

# $a1 - index of element 1

# $a2 - index of elelemnt 2

# Side Effects: Array is changed to swap element 1 and 2

Swap:

sll $t0, $a1, 2 # calcualate address of element 1

add $t0, $a0, $t0

sll $t1, $a2, 2 # calculate address of element 2

add $t1, $a0, $t1

lw $t2, 0($t0) #swap elements

lw $t3, 0($t1)

sw $t2, 0($t1)

sw $t3, 0($t0)

jr $ra

# Subprogram: PrintIntArray

# Purpose: print an array of ints

# inputs: $a0 - the base address of the array

# $a1 - the size of the array

#

PrintIntArray:

addi $sp, $sp, -16 # Stack record

sw $ra, 0($sp)

sw $s0, 4($sp)

sw $s1, 8($sp)

sw $s2, 12($sp)

move $s0, $a0 # save the base of the array to $s0

# initialization for counter loop

# $s1 is the ending index of the loop

# $s2 is the loop counter

move $s1, $a1

move $s2, $zero

la $a0 open\_bracket # print open bracket

jal PrintString

loop:

# check ending condition

sge $t0, $s2, $s1

bnez $t0, end\_loop

sll $t0, $s2, 2 # Multiply the loop counter by

# by 4 to get offset (each element

# is 4 big).

add $t0, $t0, $s0 # address of next array element

lw $a1, 0($t0) # Next array element

la $a0, comma

jal PrintInt # print the integer from array

addi $s2, $s2, 1 #increment $s0

b loop

end\_loop:

li $v0, 4 # print close bracket

la $a0, close\_bracket

syscall

lw $ra, 0($sp)

lw $s0, 4($sp)

lw $s1, 8($sp)

lw $s2, 12($sp) # restore stack and return

addi $sp, $sp, 16

jr $ra

.data

open\_bracket: .asciiz "["

close\_bracket: .asciiz "]"

comma: .asciiz ","

.include "utils.asm"

### Task 03. Time: 30 Minutes

Write a function to find and replace the values from a string