**CS2100 Computer Organisation**

***Remember to bring this along to your lab!***

**Lab #4: Making Function Calls**

**(2nd and 5th March 2020)**

[ This document is available on LumiNUS and module website <http://www.comp.nus.edu.sg/~cs2100> ]

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Student No.: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lab Group: \_\_\_\_\_

###### Notes

* You should prepare your program **before the lab**
* You may email your program to your TA
  + Please email your program **before the end of your lab session**

###### Objective

In this lab, you will use **QtSpim** to explore the idea of function calls in MIPS Assembly Code. This document and its associated files (**sayHi.asm** and **arrayFunction.asm**) can be downloaded from LumiNUS Files or the CS2100 module website.

**Task 1: Getting started (sayHi.asm) [5 marks]**

Just like any high level programming language, modularization (separating code into well-defined procedures/functions) is an important idea for assembly programming. Conceptually, making function call is actually simple: we need to "*jump*" to another portion of code (the function body) then start executing the instructions in the function body. When we reach the end of that function, another "*jump*" is needed to go back to the caller.

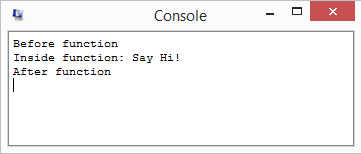
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **0x1000** | #some instruction |  | **0x3024** | #Body of Function **F** |
| **0x1004** | #Jump to Function **F** |  | **0x3028** | #Body of Function **F** |
| **0x1008** | #Continue after **F** |  | **0x302C** | #Done! Jump back to caller |
| **0x100C** | #some instruction |  |  |  |
| … | …….…. |  |  |  |

So, the simplest kind of function call can be accomplished by just two "*jump*" instructions! To facilitate function calls, MIPS gives us two variants of the "**j**" instructions, the "**jal**" (jump-and-link) and the "**jr**" (jump by register). Don’t worry, they are much easier than the name suggested.

First, download and load the assembly program "**sayHi.asm**" in QtSpim. The original content of the file is given on the next page.

|  |
| --- |
| **# sayHi.asm**  **.data**  **str1: .asciiz "Before function\n"**  **str2: .asciiz "After function\n"**  **str3: .asciiz "Inside function: Say Hi!\n"**  **.text**  **main:**  **li $v0, 4 # system call code for print\_string**  **la $a0, str1 # address of string to print**  **syscall # print the string**  **jal sayHi # Make a function call to sayHi()**  **li $v0, 4 # system call code for print\_string**  **la $a0, str2 # address of string to print**  **syscall # print the string**  **# End of main, make a syscall to "exit"**  **li $v0, 10 # system call code for exit**  **syscall # terminate program**  **# start of function sayHi()**  **sayHi:**  **li $v0, 4 # system call code for print\_string**  **la $a0, str3 # address of string to print**  **syscall # print the string**  **# Use "jr" to go back to caller** |

The intention of the program is to print 3 messages in the following order:



The first and third messages are printed in the "**main**" function while the second message is printed by the "**sayHi**" function. The given program is almost complete, with only one missing instruction. The purpose of this code is to demonstrate the necessary instructions needed for a making a function call.

Now, let us step through the program to make several observations. Use the “Single Step” button or press **F10** to go through the program line by line. Stop when you reach the instruction "**jal sayHi**".

Answer: The instruction address of "**jal sayHi**" is at **0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Press **F10** one more time to execute the "**jal**" instruction. Answer the following:

Answer: The **PC** is now at **0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Answer: The register **$31** now contains **0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

At this point, you should be able to see why the name of register **$31** is **$ra** (return address). Express the content of register **$31** with respect to the instruction address of the corresponding "**jal**" instruction. Use the notation **Addr(jal)** to indicate the instruction address of "**jal**" instruction.

Answer: **$31** = **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

If you continue stepping through, we will reach the end of the "**sayHi**" function and get ‘stuck’. We need a way to go back to the main function **and continue from where we left off**. We can do this easily by the "**jr**" (jump by register) instruction which is missing in the program. This "**jr**" instruction takes a **register number** as operand. It will jump to the address stored in the specified register. For example,

**jr $15**

The content of register **$15** will be used as the target address. This is known as **direct addressing** (the address is directly specified in full).

What is the correct register number to be used in the "**jr**" instruction so that we can jump back to main?

Answer: **jr** **\_\_\_\_\_\_**

Now, edit your code and insert the "**jr**" instruction accordingly. Run your program, you should see the 3 messages in the same order as shown in the earlier output screenshot.

**Task 2: Let's share information (arrayFunction.asm) [15 marks]**

We can now turn to other aspects of function call, namely function parameters (arguments) and function return value. Actually, we have encountered this idea in previous labs. Take a look at this very familiar sequence of using the system call **read\_int**:

|  |
| --- |
| **li $v0, 5 # System call code for read\_int**  **syscall**  **sw $v0, 0($t1) # "return result" is in $v0** |

You can see that there is an agreement to use the register **$v0** to store the system call code for the system call (a special kind of function call). Additionally, the return result (an integer read from user) is placed in register **$v0** when the system call is completed.

|  |
| --- |
| **Key idea: we can pass information to the function by placing values in registers and retrieve the return result in the same way.** |

Let us first attempt to pass information to a function. Download and load the **arrayFunction.asm** in QtSpim. The main function code is given below:

|  |
| --- |
| **.data**  **array: .word 8, 2, 1, 6, 9, 7, 3, 5, 0, 4**  **newl: .asciiz "\n"**  **.text**  **main:**  **# Print the original content of array**  **# setup the parameter(s)**  **# call the printArray function**  **# Ask the user for two indices**  **li $v0, 5 # System call code for read\_int**  **syscall**  **addi $t0, $v0, 0 # first user input in $t0**    **li $v0, 5 # System call code for read\_int**  **syscall**  **addi $t1, $v0, 0 # second user input in $t1**  **# Call the findMin function**  **# setup the parameter(s)**  **# call the function**  **# Print the min item**  *<code not shown>*  **# Calculate and print the index of min item**  *<code not shown>*    **# End of main, make a syscall to "exit"**  **li $v0, 10 # system call code for exit**  **syscall # terminate program** |

The basic flow of the program is as follows:

1. Print the original content of array.
2. Ask the user for two indices **X** and **Y**, where X ≤ Y.
3. Find the minimum item between A[X] and A[Y] (inclusive).
4. Print the minimum item and the index of the minimum item.

You’ll need to code for parts 1, 3 and 4. Again, don’t panic as most of the code are already written! For part 1, the following function is already given in the program:

|  |
| --- |
| **### Function printArray ###**  **# Input: Array Address in $a0, Number of elements in $a1**  **# Output: None**  **# Purpose: Print array elements**  **# Registers used: $t0, $t1, $t2**  **# Assumption: Array element is word size (4-byte)**  **printArray:**  **addi $t1, $a0, 0 # $t1 is the pointer to the item**  **sll $t2, $a1, 2 # $t2 is the offset beyond the last item**  **add $t2, $a0, $t2 # $t2 is pointing beyond the last item**  **loop:**  **beq $t1, $t2, end**  **lw $t3, 0($t1) # $t3 is the current item**  **li $v0, 1 # system call code for print\_int**  **addi $a0, $t3, 0 # integer to print**  **syscall # print it**  **addi $t1, $t1, 4**  **j loop # Another iteration**  **end:**  **li $v0, 4 # system call code for print\_string**  **la $a0, newl #**  **syscall # print newline**  **jr $ra # return from this function** |

The comments at the beginning of the function give you a good idea of how to make use of this function. Pay special attention to the “input” information, which tells you where to place the expected parameters. **Without** changing this function, complete the first part of the main program. You only need to place the correct information in the registers **$a0** and **$a1** then make a function call. Test your program, and you should see the original content of array printed on screen. (Hint: Don’t forget the use of "**li**" and "**la**" instructions).

Now, let’s tackle something slightly more challenging. Let us now write a function to find the minimum element. The function header is given in the program as follows:

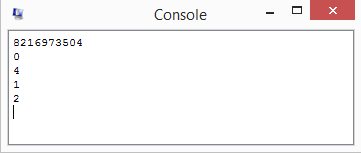
|  |
| --- |
| **##################################################################**  **### Function findMin ###**  **# Input: Lower Array Pointer in $a0, Higher Array Pointer in $a1**  **# Output: $v0 contains the address of minimum item**  **# Purpose: Find and return the minimum item**  **# between $a0 and $a1 (inclusive)**  **# Registers used: <Fill in with your register usage>**  **# Assumption: Array element is word size (4-byte), $a0 <= $a1**  **findMin:**  **# Your implementation here**  **jr $ra # return from this function** |

Note that the function expects **two addresses,** i.e. the addresses of A[X] and A[Y] in registers **$a0** and **$a1** respectively. Once the minimum item is found, the **address** of the minimum item is returned to the caller. You are supposed to use the **array pointer** approach (Lecture #8, Slide 34 & Tutorial #3, Q1b) to implement the ***findMin*** function.

Once you have written the findMin function, you are left with the last piece of puzzle to solve. How do you find out the **index** of an item from the **address** of the item? (Hint: think about how we calculate the address of an item given the index of the item.)

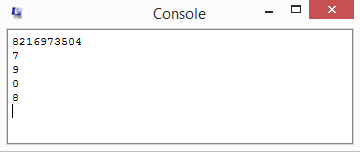
Complete the main function by calling the ***findMin*** function. Then print both the minimum item and the index of the minimum item.

Below are two separate test runs (user input circled):



**Minimum item between A[0] to A[4], i.e. {8, 2, 1, 6, 9}**

* **Minimum item is “1”**
* **Minimum item is at index 2**



**Minimum item between A[7] to A[9], i.e. {5, 0, 4}**

* **Minimum item is “0”**
* **Minimum item is at index 8**

Your lab TA will test your program with some test cases.

Total marks: 20.