CS2100 Computer Organisation

Lab 05: Function Calls in MIPS (Week 28th September)

Instruction

Short and clean We have separated the lab information into i) **instruction** and ii) **report**. Whenever there is a question in the instruction (easily identified as they have **[X pts]** tagged to the end), write / type your answer in the corresponding location in the **report** document. Please take note of the submission specification at the end of this document.

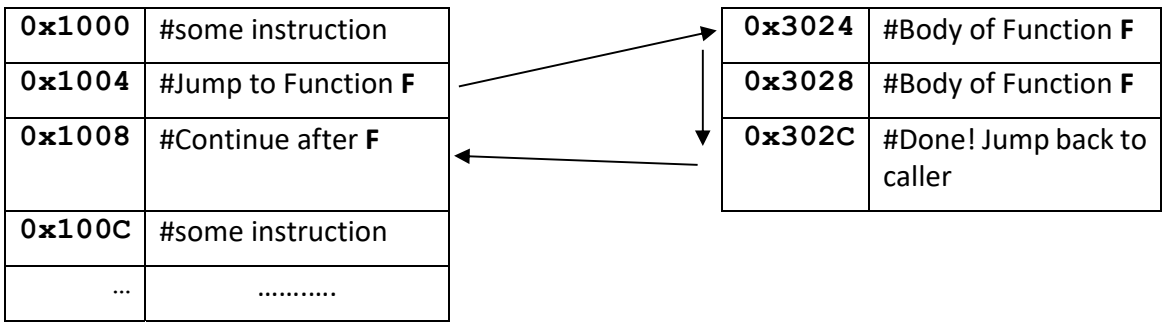
Objective

In this lab, you will use **QtSpim** to explore the idea of function calls in MIPS Assembly Code. Make sure you have access to all the files included in the **`Lab05.zip`** archive: **`sayHi.asm`** and **`arrayFunction.asm`**.

Please note that this lab is challenging (a bit more tougher than lab 04). Complete as much as you can before attending the lab session. It is unlikely that you would be able to finish in time otherwise.

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.

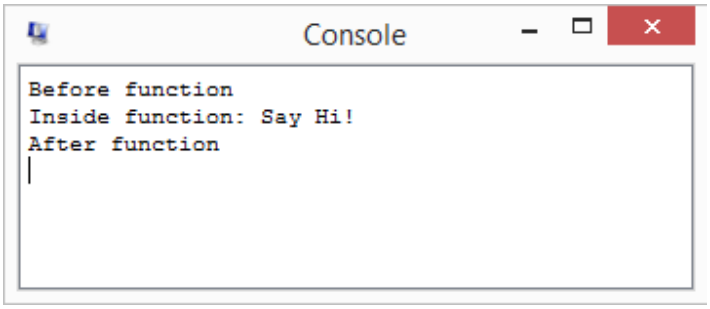


So, the simplest kind of function call can be accomplished by just two "*jump*" instructions! To facilitate function calls, MIPS gives us two variants of the "**j**" instructions, the "**jal**" (jump-and-link) and the "**jr**" (jump by register). Don't worry, they are much easier than the name suggested.

First, load the assembly program "**sayHi.asm**" in QtSpim. The original content of the file is shown below:

```
# 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:

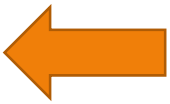


The first and third messages are printed in the "**main**" function while the second message is printed by the "**sayHi**" function. The given program is almost complete, with only one missing instruction. The purpose of this code is to demonstrate the necessary instructions needed for a making a function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

Now, let us step through the program to make several observations:

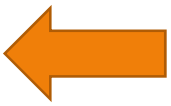
i. Use the "Single Step" button or press **F10** to go through the program line by line. Stop when you reach the instruction "**jal sayHi**".

   Give the instruction address of "**jal sayHi**" **[1pt]**

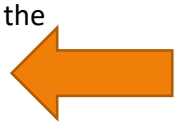


ii. Press **F10** one more time to execute the "**jal**" instruction. After the instruction, give the following values in the report:

   a) **PC** and b) Content of register **$31** **[1pt each]**



iii. At this point, you can probably 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. **[1pt]**

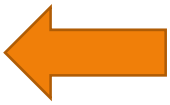


If you continue to step through the execution, we will reach the end of the "**sayHi**" function and get 'stuck'. We need a way to go back to the main function **and continue from where we left off**. We can do this easily by the "**jr**" (jump by register) instruction which is missing in the program. 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).

Give the correct "**jr**" instruction so that we can jump back to main. **[1pt]**



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 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:
    # 1. Setup the parameter(s)
    # Call the printArray function to print the original content

    # 2. 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

    # 3. Setup the parameter(s)
    # Call the reverseArray function

    # 4. Setup the parameter(s)
    # Call the printArray function to print the modified array

    # 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. Reverse the array items between A[X] and A[Y] (inclusive).
4. Print the modified content of array.

You'll need to code for **<steps>** 1, 3 and 4. Don't panic! Basic skeleton of the code is provided to guide you. For <**step 1>**, the following function is 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 reverse the items in the array. The function header is given in the program as follows:

```
#########################################################################
###   Function reverseArray                                           ###
# Input: Array Address in $a0, Number of elements in $a1
# Output: NIL, as the array items are modified directly
# Purpose: Reverse the array items, array pointer approach must be
#          used.
# Registers used: <Fill in with your register usage>
# Assumption: Array element is word size (4-byte)
#             $a0 is valid and $a1 is positive integer
reverseArray:
      # Your implementation here
      jr $ra                  # return from this function
```

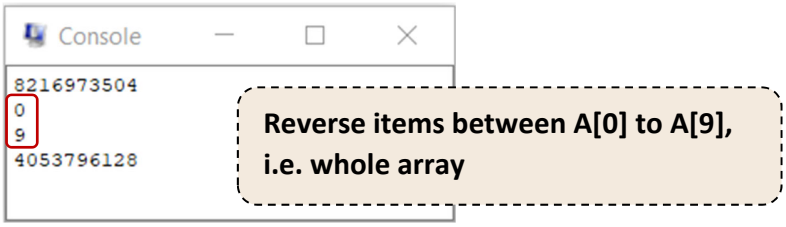
Please note that **you are not allowed to change the function design, i.e. the input / output restrictions must be respected**.

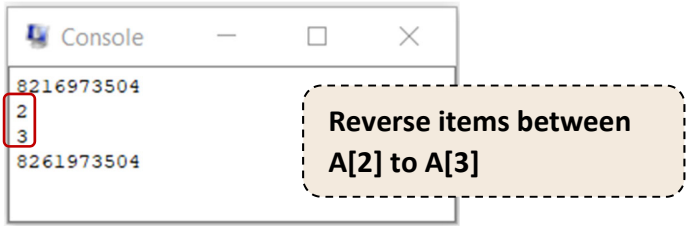
You should notice several challenges right away:

a. The `reverseArray` function takes in a single array pointer and the number of elements only. You need to figure out how to make this function reverse **a certain part of the array** only. [Hint: You learned something like this in lecture set 08 ….]
b. The implementation requires the **array pointer** approach, i.e. instead of using index to calculate the array element address, you should maintain a number of memory pointer(s) in order to manipulate the items instead.

Complete the main function by **setting up the parameters** and call the `reverseArray` function. Print the updated array afterwards.

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





Demonstrate the finished program to your lab TA. Your lab TA will test your program with a number of inputs. **[9 pts]**

You do not need to submit any program.
Please do not send your programs to your labTA after the lab; they will not be accepted.

Submission
You need to submit into Luminus Folder no longer than **2359 on the same day** you have the online lab. Please submit the **report document** as **[student_id]_lab05.doc (or PDF)**, e.g. **A1234567X_lab05.pdf**. Submit into the correct folder on Luminus before 2359 of your lab day.

Marking Scheme: Report & Code – 5 marks; Demonstration – 9 marks; Total: 14 marks.