CSCB58 Lab 5

Part1: Arrays and Functions

Introduction

Last week, we built branches and loops in MIPS assembly using labels and branches. This week, we will write some programs with arrays and functions, and practice using syscalls. You may also find these techniques useful for your project.

This week's lab assumes that you completed all the material from last week's lab. If you did not get the programs working last week, finish them before this lab.

[TASK] Document your code with comments, and write your name at the top of your submissions.

[TASK] Do not forget to terminate the program correctly using syscall 10.

Input and Output Using Syscalls

Syscalls can be used to get input from and output to the user (see lecture). The following program writes "Hello!", reads a number from the user, and outputs the number plus 1.

```
.data
str:    .asciiz   "Hello!\n"

.text
.globl main
main:
        # Print "Hello!" (address of string to print should be in $a0)
        li $v0, 4
        la $a0, str
        syscall

        # Read a number (result will be in $v0)
        li $v0, 5
        syscall

        addi $t0, $v0, 1    # $t0 = $v0 + 1

        # Print result
        li $v0, 1
        move $a0, $t0       # number to print should be in $a0
        syscall

        # End program
        li $v0, 10
        syscall
```

Write a simple assembly that does the following:

1. Print "Enter 3 numbers: a, b and c". Don't forget to add a newline \n at the end.
2. Read a number from the user (let's call this number a).
3. Read another number (let's call this number b).
4. Read a third number (let's call this number c).
5. Compute the number of of solutions to the equation $ax^2 + bx + c$. See previous lab for how to do that. You can use the same code if you want.
6. Write the following output "There are **N** solutions for ax^2+bx+c" where **N** is replaced by the number of solutions (there is no need to replace a, b, c with the numbers). **Hint:** you will need to combine multiple syscalls to do this! Think about when you should or shouldn't include \n.

[TASK] Save your code as `lab05P1a.asm`, zip it with the other programs and submit to Quercus.

Arrays

As we discussed in the lecture, arrays are declared in the `.data` section of the assembly code. For example, the following code fragment declares two arrays of 7 integers. The first one is initialize to some numbers, and the second one has 7 integers all initialized to 0. The variable LEN stores the length of the arrays.

```
.data
A:    .word    5, 8, -3, 4, -7, 2, 33
B:    .word    0:7
LEN:  .word    7  # length of the arrays
```

To access the elements of an array, we need to perform memory access instructions such as `lw` and `sw`. To access an element we must first compute its memory address. The basic formula to compute the memory address of an element is `base + offset`, where `base` is the address of the first element of the array (the label `A`), and `offset` is the index of the element multiplied by the size in bytes of each element of the array. Index of arrays is zero-based: it is always between 0 and array length minus 1. For example, the value of A[4] is -7, and the memory address for A[4] is `A + 4*4`. See the lectures for more details.

Write a program code that iterates through A, multiplies each element by a factor that depends on whether the number is odd or even, then stores the result in the equivalent element of B (hint: you can use `andi` with 1 to check if an element is odd). Your program should take the length of the arrays from the variable LEN, and should work for any length of an array. Even numbers in A are multiplied by 10, while odd numbers in A are multiplied by 5.
In Python this would be something like: `B[i] = A[i] * (10 if (A[i] % 2) == 0 else 5)`.

[TASK] Save your code as `lab05P1b.asm` and submit to Quercus with the other programs in a zip file.

Function Calls

Just like any high level programming language, separating code into well defined procedures or functions is an important idea. 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.

Passing arguments and return values can be done in many different ways, depending on the *calling convention*. In this lab we are going to use two different calling convention: the simple stack-based calling convention we learned in the lecture, and the register-based calling convention where arguments are passed using `$a0` to `$a3` (first argument in `$a0`, second in `$a1` and so on) and return value is passed using `$v0`.

Write an assembly program to implement the pseudocode listed below, subject to the following requirements:

- The function `do_addition` should use the register-based calling convention.
- The function `n_solutions` should use the stack-based calling convention.
- You will need to explain parts of your code to your TA and convince them you are using the correct calling convention.
- We are not going to be strict about whitespace as long as you achieve the needed important goals. If in doubt, follow the Python code.

[TASK] Save your code as `lab05P1c.asm` and submit to Quercus with the other programs in a zip file.

The code (in Python) you need to implement in assembly:

```
# in main
A = input("Enter a value for A")    # read A
B = input("Enter a value for B")    # read B
C = input("Enter a value for C")    # read C
print("Before function")            # this might be useful when debugging
print("A + B + C = ", do_addition(A, B, C))
print("Num solutions for Ax^2 + Bx + C is = ", n_solutions(A, B, C))

# Add the numbers
def do_addition(A, B, C):
    return A + B + C

# Compute number of solutions to quadratic
def n_solutions(A, B, C):
    delta = B*B - 4*A*C
    if delta > 0:
        return 2
    if delta == 0:
        return 1
    return 0
```

Part 2: Recursion

Last week, you wrote some programs that use basic functions and arrays. This week, you'll write a program that puts this all together: a recursive function that finds an element in a sorted array using binary search.

[TASK] Document your code with comments, and write your name at the top of your submissions!

Binary Search

Implement a recursive function `binsearch` that performs **binary search** on a reverse sorted array, all in assembly. **No loops allowed!** You will also implement a program to call this function.

The function you will implement would looks like this in C (i.e., this is its signature):

```
int binsearch(int arr[], int n, int x)
```

The function `binsearch` accepts 3 parameters: an array of integers `arr` that is sorted in descending order (from highest to lowest), the number of elements in the array `n`, and a number to find in the array `x`. `binsearch` performs a recursive binary search to find `x` in the array in logarithmic time (in other words, $O(\log n)$). If `x` is found, the function will return the zero-based index of the position of `x` in `arr`. If `x` is not found, the function will return -1.

Requirements:

- Use the usual (simple) stack calling convention, but remember to preserve registers as needed.
- Do not change the function signature or behaviour of `binsearch`. It must behave exactly as described.
- **Loops are not allowed**! You will need to convince your TA that your code is recursive and not using loops.
- You can assume that the input is valid: `arr` is sorted in descending order, `n` is zero or positive, and will never be larger than the length of `arr`.
- You may also assume that all elements in `arr` are unique (no number appears twice).
- Your function needs to work on any valid input, not just the one we ask you below. Test it well!

Guidance

As with our previous lab, you will write an assembly program that implements `binsearch` and calls it from `main`. Your program should include the following variables.

```
.data
array:   .word   101, 100, 65, 8, 4, 1, -5, -30
length:  .word   8
```

The program will

1. Print `"Enter value to find:\n"`
2. Read a number `val` from the user
3. Call `binsearch(array, length, val)` and get the return value.
4. Print `"Result is:\n"`
5. Print the result (the return value).

[TASK] Save your program as `lab5P2.asm` and submit to Quercus. Be prepared to explain your code.

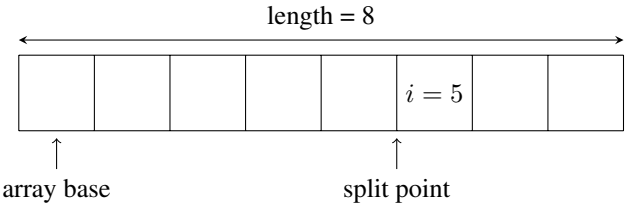
Here is an example of running the program:

```
Enter value to find:
8
Result is:
3
```

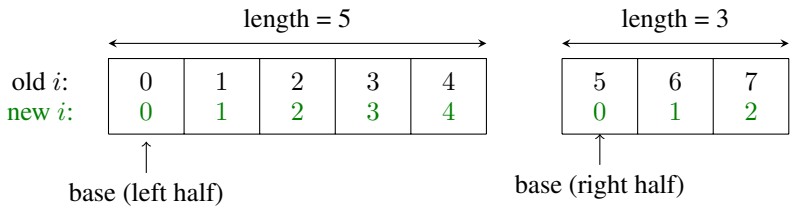
Tips and Hints

This is challenging. Expect to spend a few hours with MARS/Python. An implementation of `binserach` takes something in in the area of 100 lines of code (this includes assembly, comments, and empty line).

- Use recursive solution. Start small, and solve the problem at a high level, maybe on paper. It's not a very hard problem!
- There are a lot of edge cases! First implement a recursive function in a high-level language (Python or C). Test it well, and proceed to assembly once you are sure it is correct. Once you have a working implementation in a high-level language that has all the detail, converting to assembly is not very hard.
- Don't even try to implement it without testing and debugging in MARS. We are not asking you to do this on paper or in your head.
- Test your assembly version too! Test it really well on edge cases and various inputs.
- You will need to slice (split) an array to two parts. How to do this? Remember that as far as the function is concerned, an array is just a base address and a length.



You efficiently split an array to two parts at index i by: (a) pretending it is shorter for the left half, and (b) for the right half by treating address of index i as the beginning (base address) of the new, smaller half (as well as also pretending it is shorter). No copying is needed – everything stays in the same place in memory. Note that what used to be index i in the original array is now at index 0 in the right array, so you may have to adjust return values in your recursion.



(This trick works in C as well! Python has its own mechanism for slicing.)

- Don't forget to save registers when you need to. Forgetting to save a register will probably happen, and will cause bugs.
- Program crashes are often caused by a bad jump or by executing an invalid instruction (meaning machine code that does not get decoded to anything valid). The root cause could be messing up the stack or `$ra`. Another cause for crashes is misaligned memory access.

Evaluation

As always, marks are based not only on submitted work but also on oral examination by TA. You need to be able to make reasonable explanations about any details to the TA.

Solution	Submitting everything on time	1 mark
	Part 1:	
	Syscalls	1 mark
	Arrays	1 mark
	Functions	2 marks
	Part 2:	
	Main program	1 mark
	Recursive binsearch	3 marks
Understanding	TA oral score	1 to 4

Final lab marks (up to 9) are determined by multiplying your solution subtotal by the oral score (1–4) and dividing by 4:

$$\text{total} = \text{solution marks} \times \frac{\text{oral score}}{4}$$