# Laboratory Exercise 7

Subroutines in RISC-V assembly

Revision of November 4, 2025

In this lab, you will learn how to make a subroutine in the RISC-V assembly language. You will learn the NIOS V subroutine calling convention, as well as how transfer of control works and parameter passing.

## 1 Part I

In this part, you are given a pre-written program to count the longest string of 1s in a word of data. This is the same code that was provided in Lab 6.

```
# Program that counts consecutive 1's.
.global _start
.text

_start:
        la s2, LIST         # Load the memory address into s2
        lw s3, 0(s2)
        addi s4, zero, 0    # Register s4 will hold the result

LOOP:
        beqz s3, END        # Loop until data contains no more 1's
        srli s2, s3, 1      # Perform SHIFT, followed by AND
        and s3, s3, s2
        addi s4, s4, 1      # Count the string lengths so far
        j LOOP

END: j END

 .global LIST
.data
 LIST:
.word 0x103fe00f
```

You are to modify this code to count the longest string of 1s from a *list* of words. To do so, you must first modify the given code to use subroutines.

Your subroutine must follow proper calling conventions as described below.

**The NIOS V Subroutine Calling Convention**

As discussed in class, the NIOS V Subroutine Calling Convention provides guidelines ("rules") for how registers should be used in a program, especially when subroutines are involved. The convention is as follows:

Register `t0` to `t6` are caller saved registers meaning that it is the job of the caller to save these registers on the stack if the caller needs them after the subroutine returns to the caller.

Registers `s0` to `s11` are callee saved registers meaning that it is the job of the subroutine itself to save these registers on the stack at the beginning of the subroutine if they are to be used.

The proper procedure for passing parameters to a subroutine is to use registers `a0` to `a7`. If there are more than 8 parameters that have to be passed to a subroutine they must be saved in the stack in reverse order. For example, if we have 10 parameters, we would save parameter 10 first, and then parameter 9. Inside the subroutine we would pop parameter 9 first and then parameter 10.

In order to return a value from a subroutine, the `a0` and `a1` registers are used. If more results are to be returned, then the stack is used.

(Note: if you are using CPUlator, then you will see an error stating that you have "clobbered" one or more of the registers `s0` to `s11` if you over-write these registers in a subroutine without saving/restoring their values as required by the calling convention.)

**You should start using this calling convention** for all NIOS V code that you write in this lab, in all subsequent lab exercises, and on all test/exam questions in this course.

## What you should do

**Pre-lab work (to be completed before coming to lab):**

1. Create a new RISC-V assembly language program. Start by typing the RISC-V assembly language code shown above into CPUlator. Make sure you understand how this code works.
2. Make a copy of the created file and call it part1.s.
3. Take the code that calculates the number of consecutive 1s and make it into a subroutine called `ONES`. The `ONES` subroutine must use register `a0` for passing the input and returning the result. Make sure your subroutine follows proper calling convention.

**In-lab work:**

4. Add more words in memory starting from the label `LIST`. You can add as many words as you like — but include at least 10 words. Terminate the list with a `-1`.
5. In your main routine, call the newly created subroutine in a loop for every word in the list. Count the number of 1s for every word and keep track of the largest number of 1s.

Store this result in register s10 when your program completes execution.

6. Make sure to use the single-step capability of the simulator to step through your code and observe what happens when your subroutine call and return instructions are executed.
7. Once you are satisfied with the results, submit your code to the automarker; make sure your code passes the tester before submission.

## 2 Part II

In this part, you will write a program to sort a list of 32-bit unsigned numbers in ascending order. The first 32-bit number in the list gives the number of items in the list and the remainder of the entries are the numbers to be sorted. The list your program will sort, including the number count, can be defined using the .word directive at the end of your program as follows:

```
LIST:
.word 10, 1400, 45, 23, 5, 3, 8, 17, 4, 20, 33
```

Since the first number in the list is 10, we know that there are 10 numbers to be sorted. You must use **bubble sort** to perform the sorting. To start with, write pseudo code – similar to C or Python code – to show how your sorting algorithm works. Your TA may ask to see this to determine if you are using the right approach for sorting.

In assembly, you should have a main routine that loops through the list to sort it. The list of data must be sorted "in place", meaning that you are not allowed to create a copy in memory of the list to do the sorting. As your main routine loops through the list, you will need a subroutine to compare two list entries and if needed, swap them. To do this, you must write a SWAP subroutine. The SWAP subroutine is passed the address of a list element, compares it to the following element in the list, and swaps the two elements in memory if necessary. The SWAP subroutine should return 1 if a swap is performed, and 0 if not. Pass the address of the first list element to SWAP in register a0 and pass the return value back to the main program in register a0.

Your main program (_start:) and your subroutine (SWAP) should be submitted as one file called part2.s.

### What you should do

**Pre-lab work (to be completed before coming to lab):**

1. Write pseudo code for your bubble sort algorithm

**In-lab work:**

2. Implement a bubble sort algorithm in assembly using the SWAP subroutine, and save your

code as `part2.s`

3. In CPUlator, use the disassembly pane to identify the memory address of `LIST`.
4. After running your program, navigate to the memory address of `LIST` in the CPUlator memory pane to verify that the list has been sorted correctly.
5. If your code does not work as expected, use the step-through features in CPUlator to debug it.
6. Once you are satisfied with the results, submit your code to the automarker; make sure your code passes the tester before submission.

## 3 Submission

Please submit part1.s and part2.s. An outline of the program for Part I and II is shown below, along with submission requirements.

```
.global _start
.text

_start:
# Write your code here

END: j END

.global LIST
.data
LIST:
.word #, #, #, ..., #
```

1. The code containing the input data (i.e., starting with `.global LIST` and ending with the last number) must be at the end of the file.
2. The name of the list must be `LIST`.
3. The last instruction of your program (i.e., `END: j END`) must be at the end of the `.text` section.
4. Make sure that you save the correct values to the appropriate registers, as specified in Part 1 and Part 2.