# Week 9 Lab RISC-V Assembly language programming

## Objectives

Develop understanding and experience of:

1. Using labels in assembly language code as a way to indicate a memory address.
2. Using memory and outputting a string to the console.
3. Using jump and return instructions to create subroutines

Please note that some of these tasks are very similar to those required in the coursework.

## Part B Using jump and return to create subroutines

What command do we use to jump to a subroutine?

jal

What command do we use to return back to the main program?

ret

Where is the return address stored?

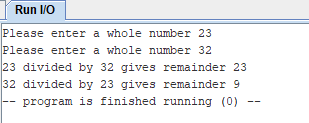
ra

The tutor will give some guidance as to how to start the next exercise which is very similar to part of the coursework. You may move ahead if you don’t need the tutor’s introduction.

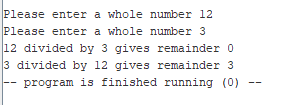
* Download the program **lab9BaStarter.asm** from Moodle.
  + Assemble and run the program. It asks the user for two numbers (integers) and gives the remainder when dividing each by the other.
  + Test the program with various inputs. This program was part of a program to find out if one number is a multiple of the other.

You should notice that the program has a lot of repeated code. Use subroutines to reduce the amount of repetition. Make sure the subroutines are placed after the code to exit the program. Note that we have not yet covered how to use a subroutine that calls another subroutine.

Screenshot of Run I/O before any editing:

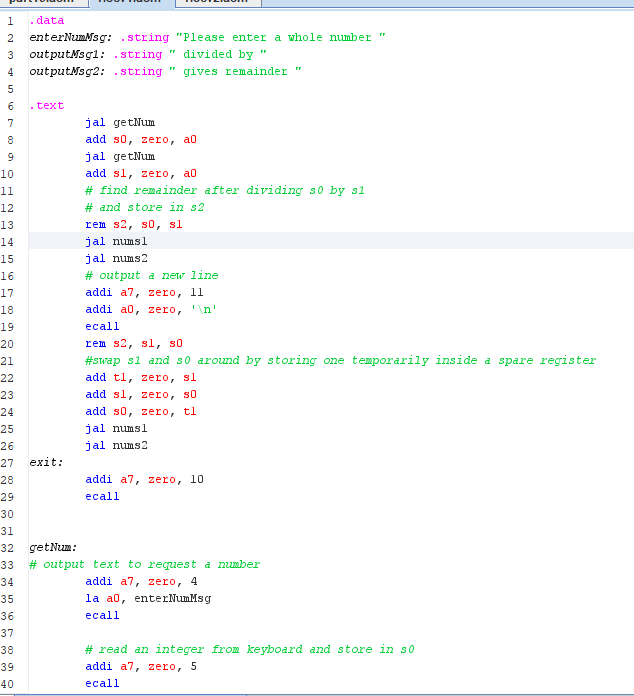


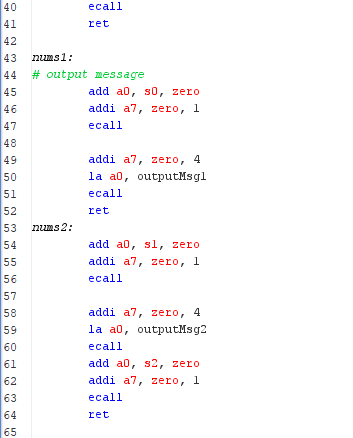
Screenshot of run I/O after editing and including subroutines:



Screenshot of code:

NOTE: There are many correct ways to do this





* Download the program week8lect\_addition.asm from the folder of selected code from the week 8 lecture.
  + Assemble and run the program.
  + Amend the program so that it outputs three separate calculations (on separate lines), using addi to store different values in the registers for each calculation.
  + Use a subroutine for the code that does the calculation and outputs to the console.
  + Make sure that you add the code to exit the program before your subroutines.

Expected Result:

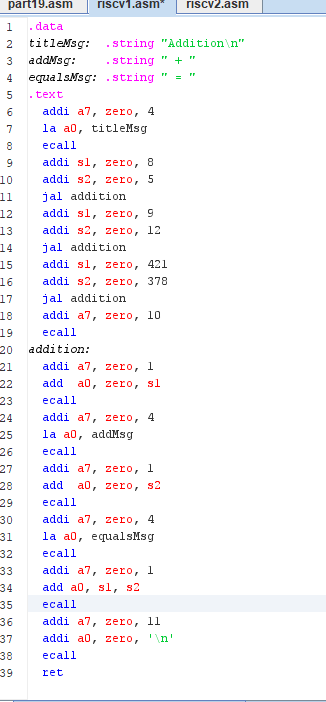
Addition

8 + 5 = 13

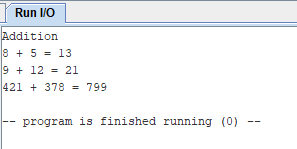
9 + 12 = 21

421 + 378 = 799

Screenshot of the code:



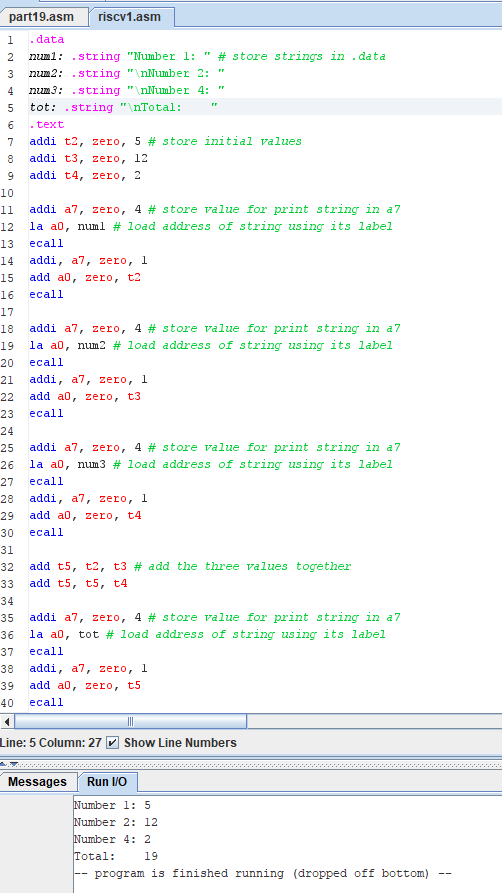
Screenshot of the Run I/O:



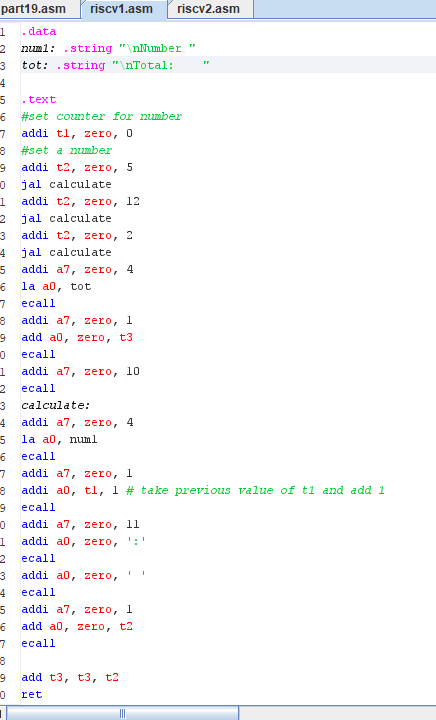
* Make a copy of your program that adds three numbers that you have improved in part A of this lab.
  + Amend the program to use subroutines.
  + Consider how you can make your program most flexible to have subroutines that could be reused.
  + You will probably want to use a register to store the number we are on instead of hard coding text “Number 1”, “Number 2” etc.

Note that because we are using registers (rather than memory) we can’t just add 1 to get from t3 to t4.

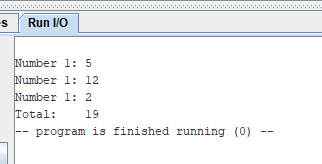
Screenshot of code prior to subroutines:



Screenshot of code after subroutines are added:

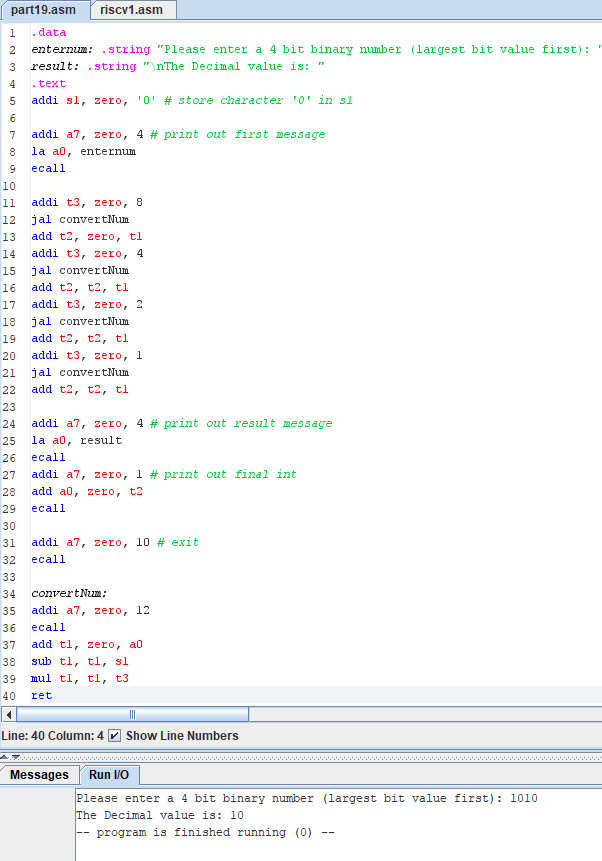


Screenshot of the Run I/O:



## Extension Exercises

1. Amend the extension exercise B (binary conversion) from Part A to use subroutines.



1. Travel calculator.

Create an application that allows the user to calculate the cost of two trips. You could imagine someone visiting some universities some distance from their home, so they have to stay overnight.

For each trip, the user should be asked for the cost per night, the number of nights and the travel cost. The program should calculate the total cost for a trip as the nightly cost multiplied by the number of nights and added to the travel cost. The total cost for the two trips should then be calculated. You may assume that all the inputs are integers.

You may wish to start by just catering for a single trip and checking that your program works. When extending to two trips, you should use subroutines to reduce the amount of repeated code when catering for the two trips. Your program does not need to keep the input details once the total has been calculated. Make sure that you check the calculations separately from your program.

You can be flexible about the wording and how the console interaction is laid out, but for an example, one run of my program appears as follows:

Trip 1

Enter the cost of each night 25

Enter the number of nights 5

Enter the travel cost 60

Total for this trip: 185

Trip 2

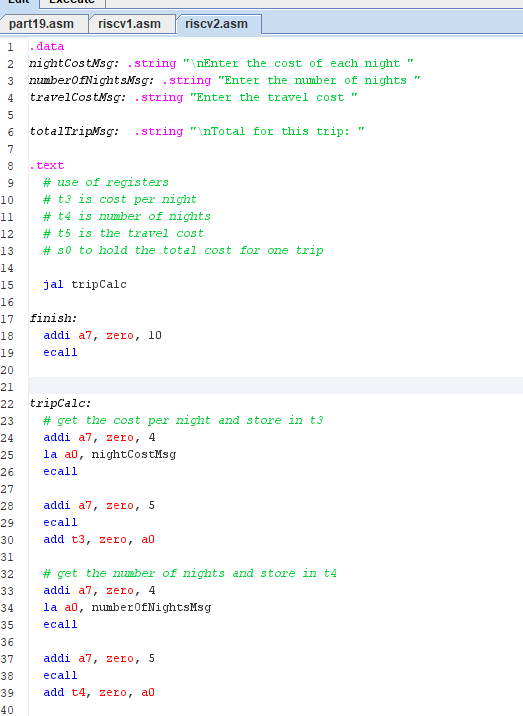
Enter the cost of each night 40

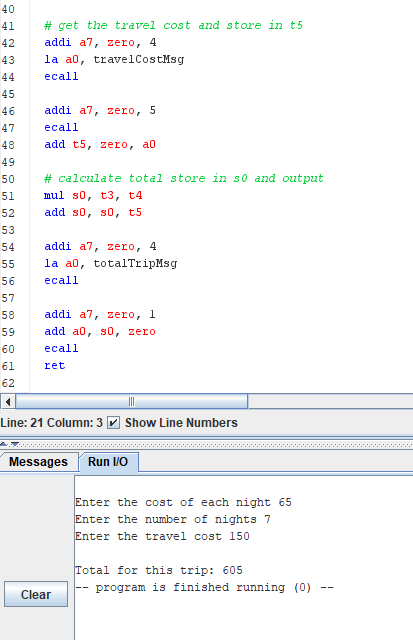
Enter the number of nights 3

Enter the travel cost 120

Total for this trip: 240

The grand total is: 425





1. Handling strings as input. Amend your travel calculator so that it also asks the user the destination of the trip, storing that information and then outputting it again when it gives the total for that trip. Again, you don’t need to store the destination of the first trip when processing the second and don’t need to handle invalid input.

For this task you will need to use the help for syscalls. When looking at inputing a string, you will see that you have to specify an address in RAM to store the string. As with output strings, this will need to be declared in the data section. One way to do so would be to create a dummy town name that will be overwritten when the input is read in. For example,

townName: .string “Unknown town “

A better way to set this up would be to use .space which specifies a number of bytes to reserve in RAM. The syntax would be

*label:* .space *nn*

where you specify the label and replace *nn* by the number of bytes you want to reserve. See the directives tab in the RARS help for a description of .string.

