

COMP15111 Lab 2 – Control Structures

1.1 Aims

To practice converting simple Python statements into RISC-V assembly code.

1.2 Learning Outcomes

On successful completion of this exercise, you will:

- have used various techniques to program control flow with RISC-V assembly.
- translate Python code to RISC-V assembly.

1.3 Summary

Create a small arithmetic calculator that performs a number of basic operations.

1.4 Deadline

Each lab exercise has the usual deadline which is the Friday of the week after the current one. All of our deadlines are listed on the unit's landing page on Blackboard. *We encourage you to engage with this assignment long before the deadline:* a) this will allow you to use the lab sessions productively, b) you will be able to manage your time better (e.g. not having 2-3 deadlines at the same time), and c) you will be able to accommodate unexpected delays. You can submit earlier if you are ready.

Remember that you must tag your commits in the usual way to show that you completed your work by the deadline. The tag for this assignment is **Lab-2-Marking**

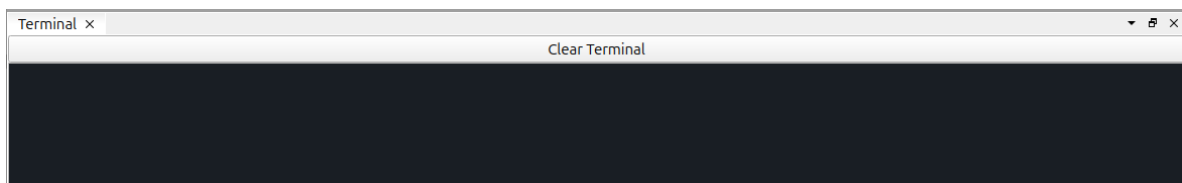
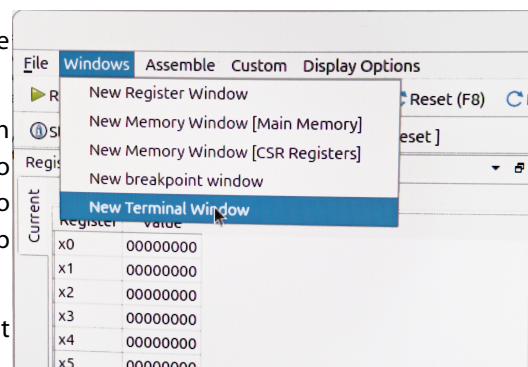
On Blackboard, there are specific instructions on how to use git and Bennett.

1.5 Description

As usual, we will use Bennett to run our programs. In case you are not familiar with Bennett, please revisit Lab 1.

Lab 2 consists of three parts, which require us to interact with Bennett by providing input and receiving output. In order to enable Input/Output functionality to Bennett, we need to enable its terminal by clicking Terminal->Start at the top option menu.

After you do that, you should see the terminal of Bennett at the bottom:



You can use the terminal, to provide input to Bennett as well as receive output from your running

programs (via the **ecall** operations). It might be the case that the background colour is white or black; it does not make any difference. You can reset the terminal by pressing “Clear Terminal” towards its top.

Part 1 – Simple Addition Program

Your copy of the file “part1.s” contains the skeleton of a RISC-V program, with some pseudo-Python code in comments. The program outputs “Operand 1: ” in the Terminal window, waits for you to type any decimal digit into the Terminal window, stores the number **corresponding** to that digit and prints it. Subsequently, it does the same for Operand 2. Finally, it prints “Result of Addition: ” followed by the sum of the two operands.

Task (3 point): Edit the program (your .s file) to insert RISC-V instructions that are a translation of the pseudo-code in the comments.

The operands and the result of the addition must be stored in memory!

Note: You can find a list of all ecall operations in the handout of Video 11 (“RISC-V Leftovers”).

If you open the .s file you will see some Python code inside the file as a comment. You will need to provide the corresponding RISC-V instructions. When a comment refers to x10, you should just use register x10, rather than a variable. Do not change any of the RISC-V instructions already in the program.

Hint 1: The assembler automatically translates character literals (e.g. ‘c’) into their ASCII code. Similarly, it converts strings (e.g. “Hello”) into a sequence of bytes in memory where each byte is the ASCII code of the corresponding character.

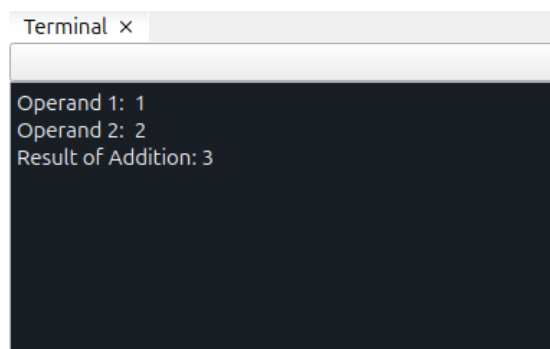
Hint 2: The “return” (enter) character, that you use to change lines is the character ‘\n’.

Hint 3: The ASCII characters for decimal numbers (‘0’-‘9’) are in a single ASCII block that starts with ‘0’ and ends with ‘9’. A character is then ASCII if its code is in the code range [‘0’, ‘9’]. Similarly, translating ASCII codes for numbers into numbers is straightforward.

Compile, load, reset and run your edited code; remember to check that there are no syntax errors. Test for these corner cases explicitly:

- What happens if the user tries to input non-digit characters, e.g. ‘a’?
- What happens if the user gives ‘0’ or ‘9’?
- What happens if the user sets both operands to 9?

If you get it right, the output should look like the one from the image below:

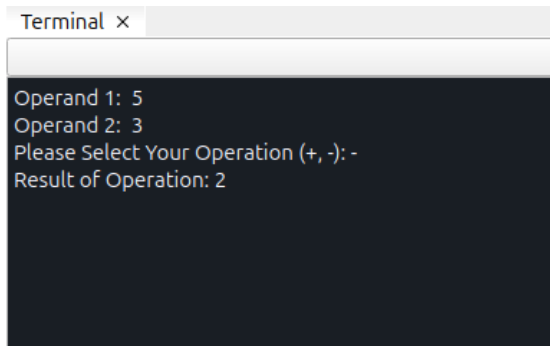


```
Terminal x
Operand 1: 1
Operand 2: 2
Result of Addition: 3
```

Part 2 – Enhancing the calculator

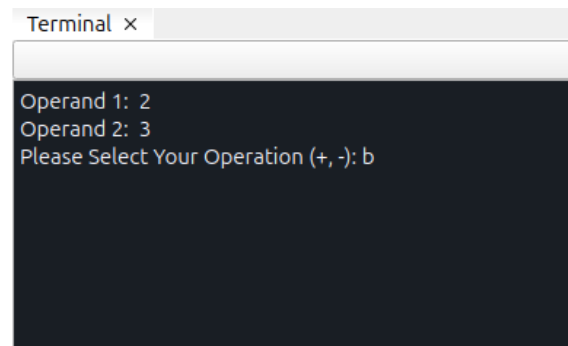
First, copy your solution in **part1.s** into a new file called **part2.s**. This will be the starting point for your work in Part 2.

Task (2 points): Edit the program to: a) allow the user to select an operator (plus or minus) to apply on the operands, and b) apply the selected operator. If the user selects an operator that does not exist, the program should exit. The output should look like the ones below:



```
Terminal x
Operand 1: 5
Operand 2: 3
Please Select Your Operation (+, -): -
Result of Operation: 2
```

Figure 1: Subtraction Example



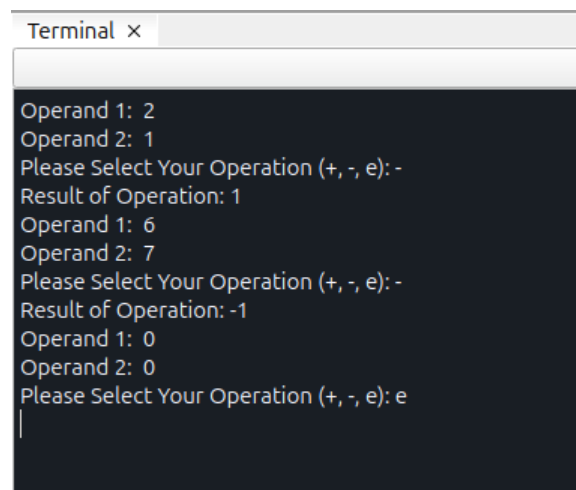
```
Terminal x
Operand 1: 2
Operand 2: 3
Please Select Your Operation (+, -): b
```

Figure 2: Invalid Operation Example

Part 3 – Continuous Operation

First, copy your solution in **part2.s** into a new file called **part3.s**. This will be the starting point for your work in Part 3.

Task (2 points): Edit the program to make it operate continuously until an exit character is entered. Turn the main body of the previous program into a loop and execute the STOP ecall only when the user asks for the operation “e”. The output of the program should look like this:



```
Terminal x
Operand 1: 2
Operand 2: 1
Please Select Your Operation (+, -, e): -
Result of Operation: 1
Operand 1: 6
Operand 2: 7
Please Select Your Operation (+, -, e): -
Result of Operation: -1
Operand 1: 0
Operand 2: 0
Please Select Your Operation (+, -, e): e
|
```

Part 4 – Optimise

Make your changes directly in **part3.s**. Your changes should not change the functionality of the program!

Task A (1 point): Edit the program to eliminate the use of temporary variables, including the ones used to store the operands and the result. Ideally, your only memory data after this should be your strings.

Task B (1 point): Edit the program to minimise the number of control flow instructions inside the loop. Anything less than seven is acceptable.

Task C (1 point): Edit the program to minimise the number of instructions executed inside the loop. Consider for example whether you need to reload some immediate values in every iteration. Each iteration should ideally take fewer than 50 steps to execute. You can calculate the number of steps you need by comparing the steps counter (under the single step button) between two consecutive iterations of the program's main loop.

The output of the program should remain the same.

1.6 Completion, Feedback and Marking Process

The total mark for Lab 2 is 10 points. As soon as you have completed the exercise, you need to git commit and push to your repository your three files. Make sure that any new files you might create are added to the repository (git add).

We will mark all submissions within 2 weeks of the deadline. We will provide detailed feedback, which you will receive by email when we mark your submission. We will also discuss this assignment in a live session after the extended submission deadline.

File(s) to be committed to Gitlab

1. part1.s,
2. part2.s and
3. part3.s.