COMP15111 Lab 4 – Methods and (more) Control Structures

1.1 Aims

To practice converting Python control structures into RISC-V assembly code and familiarise with stack operations.

1.2 Learning Outcomes

On successful completion of this exercise, you will:

- have coded a Python method in various styles using RISC-V code
- have coded Python conditional evaluation operations (**and, or**) using RISC-V code

1.3 Summary

Translate a Python program into RISC-V code. The program consists of calls to a method, that uses parameters and local variables and contains a `while` statement and an `if-then-else` statement, both of which use conditional Boolean operations (**and** and **or**).

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-4-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.

There is a file, "part1.s", that contains the Python program as a set of comments, with some RISC-V code. The RISC-V code deals with the output and a simplified version of the rest of the program:

- For part 1 you will need to complete the implementation of the printAgeHistory method, by passing arguments and saving used variables.
- For part 2 you will need efficiently use the stack to pass parameters.
- In part 3 you need to tidy up the code by abstracting some repeated sections into a method of your

own.

- The conditions in the "while" and "if" statements have not been properly translated - for part 4 you are supposed to correct this.

Part 1 – Passing parameters using registers

For this part, you should edit the file you have been given, "part1.s".
This code defines the method 'printAgeHistory'. The method is called with a **jal** instruction.
Inside the method some working registers are needed but these may already hold important values. The working registers therefore need to be stacked before they are modified (e.g. at the start of the method); they must also be restored at the end of the method, before returning to the calling code.
The stack has been set up for you at the start of the program by allocating a space in memory and pointing the Stack Pointer (sp) at an appropriate address.

Use the memory pane (bottom right) – or open a New Memory Window – to view the memory implementing the stack as you **step** through the code. Observe how the stack grows and shrinks as the code proceeds. This may help clarify your understanding of how the machine works.
The code contains some deliberate inefficiencies; for example, it is longer (and slower) than is necessary.

There is another file, "AgeHistoryPart1.py", that contains a version of the Python program with these simplified conditions – you can run this in the usual way to see how your completed RISC-V program should behave if you have done everything correctly.
Note that some of the comparisons were not implemented for part1. Have a look at the code and at the expected output of the Python file before you start coding part1!

Task (1 point): Modify the code around the method call – inside and outside.
Essentially you need to pass the parameters to the method using registers. You must follow the RISC-V ABI convention for this task. Furthermore, the method uses some other registers, that need to be protected, according to the ABI convention. You are required to save these registers, by following the callee saved approach. The parameters you need to pass are specified in the code whenever you find the comment "for part 1".
Your output should now match the output of the Python file "AgeHistoryPart1.py".

Part 2 – Passing parameters using the stack

While using registers to pass parameters could be considered an efficient way to implement method calls, when the number of arguments grows, you might have to use the stack. In this part we rely on the stack to pass parameters to the printAgeHistory method.

Task (2 points): After successfully completing the Part 1 task, copy your code from "part1.s" to a new file "part2.s" and modify it to use the stack to pass parameters. You should minimize the number of accesses to

memory. For this reason, the combined number of PUSH/POP instructions **must** not be higher than **16** in the **whole** program.

Hint: look at the ABI convention to find ways to reduce your PUSH and POP operations.

Make sure that your output still matches that of "AgeHistoryPart1.py".
In order to get full mark in this part you need to use the stack appropriately. Checks will be performed to ensure you did not corrupt any register and that pushes and pops are balanced.

Part 3 – Nesting method calls

The code which ***prints the date*** occurs (with a slight variation) in three places in the given program. This will make it more difficult to modify –e.g. if a customer wants the numbers in a different order – because that is their local convention.

Task (4 points): ***You will use your code from your part2 to complete the task***. After successfully completing the **Part 2** task, copy your code from **"part2.s"** to a new file "part3.s". Create a "***printTheDate"*** method at an appropriate place in that file. Extract the relevant code as a method in its own right; you should pass parameters using STACK only. Invoke it from the three separate places; check carefully that the correct parameters are passed each time, and you do not corrupt any values in use. The method parameters passed using Stack must be deallocated efficiently after each method call. Note that these calls will be ***nested*** inside printAgeHistory and you may need to check that you are not corrupting more of your register states. You must follow a callee-saved approach in your methods, and ONLY save to the stack what is required by the ABI and by your program to function.

Hint: if you have problems go back and step through the code observing the register states.

Part 4 – Improving the translation of the conditions

After successfully completing the Part 3 task, copy your code from "part3.s" to a new file –"part4.s" –, and use this new file to create your answer to this part.

There is another file, "AgeHistoryPart4.py", that contains the Python program with the correct conditions. You can run this in the usual way to see how your completed RISC-V program should behave if you have done everything correctly.

Again, familiarise with this code and its output before starting part 4.

Task A (1 point): Extend the function of your program by adding in the code for the extended comparisons which were commented out. These occur in two places and are marked "for part 4 . . . " in the file.
There is another file, "AgeHistoryPart4.py", that contains the Python program with the correct conditions. If you want to, you can run this in the usual way to see how your completed RISC-V program should behave if you have done everything correctly.

Task B (2 points): Remember you can optimise comparisons by taking advantage of some characteristics of Boolean operators.
E.g. in (a<b or c<d), if a is less than b, you already know the answer is true (if you forgot, you can have a look at lecture 25 again…) and there is no need to compare c and d.
Similarly, for (a<b and c<d) if a is not less than b, you already know the answer is false and you should not compare c and d.
For this task you **must** implement the enhanced checks from Part 4 Task A, using no more than **11** instructions combined.
This number is the total number of instructions for both the checks (e.g. if you implement the first check with 5 instructions and the second with 6, the total number of instructions will be 11).
Note that this number includes any instruction you use, therefore lw, sw and any arithmetic instruction are included in the count too.

The existing instructions

```
loop1    lw a0, pYear
lw a0, pMonth
```

are not included in the count.

Make sure the output of your code matches that of "AgeHistoryPart4.py".
You can submit a single solution for both these tasks, as long as the condition for both are met. These were broken down to allow you to work incrementally.

1.6 Completion, Feedback and Marking Process

The total mark for Lab 4 is 10 points. As soon as you have completed the exercise, you need to git commit and push to your repository. 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 (as part of the solution to part 1)
2) part2.s (as part of the solution to part 2)
3) part3.s (as part of the solution to part 3)
4) part4.s (as part of the solution to part 4)