FIT2085-S1-2021
Solo Prac 2

Indentation, spaces and comments in MIPS

While MIPS disregards all indentation, it is quite useful for readability. We would like you to add one level of indentation (you decide how many spaces) to quickly distinguish labels from instructions. Some people are starting to add python-like indentation to assembly languages to make clearer "then" and "else" branches, loops, etc. You do not need to do this but are welcome to do so if you want.

White spaces are also disregarded by MIPS, but they are good to distinguish among "blocks" of MIPS code, that is, sets of instructions that have a common aim. For example, the instructions required to do a syscall (reading an integer, printing a string), or a given computation (say, x = 4*y-3*z can be a block). We will not enforce a strict policy of what a block is, but it cannot be longer than the set of instructions required to implement the associated Python line.

Regarding comments, you are expected to:

- Add a header to your program with your name, the list of global variables (if any) and a short description of what the program does.
- Add a brief (a short line is often enough) comment at the beginning of each block of instructions to give a high-level idea of what it does (e.g., the associated Python line).
- Complicated or unclear statements should be commented by adding a # to its right followed by the text. This can include conditional statements (e.g. slt / beq) and any reference to the stack (e.g. -4($fp)). Perhaps not required yet, but will be required for future pracs.

As usual, you are required to use clear names for identifiers (that is, for any MIPS label you define). Once we translate functions, I will tell you how to comment functions in MIPS.

Exercise 1

Let's warm up with something simple but fundamental, defining and printing strings.

```
print("I am really enjoying MIPS")
```

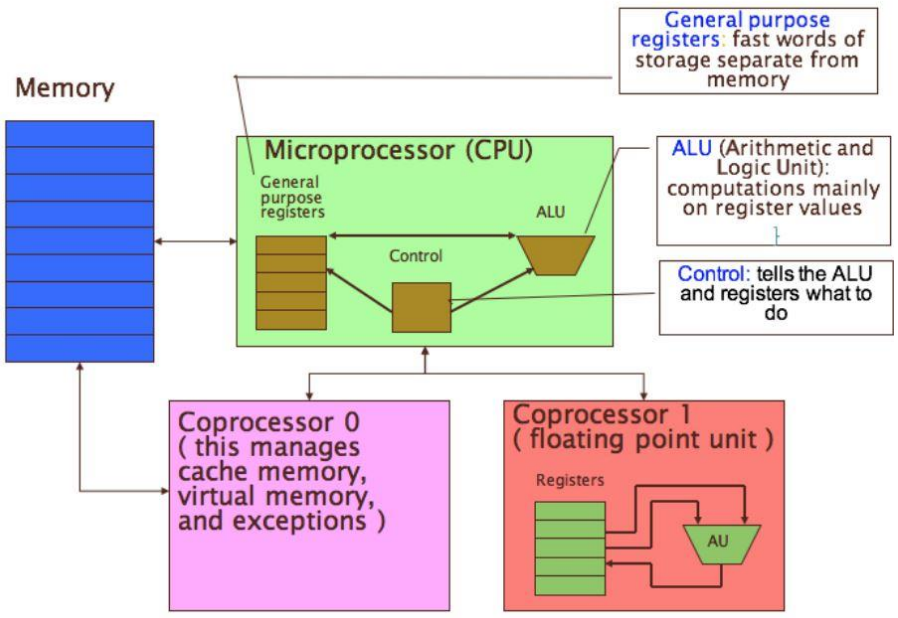
- Create a properly commented MIPS program2 in file exercise1.asm that is equivalent to the Python code. Note that the last lesson in week 1 (MIPS Programs and Instruction Set) has an example of how to declare strings using the .asciiz assembler directive, of the syscall code you need for printing a string, and of the la pseudo-instruction which you will need to load the address of the string into a register. In addition, the tutorial in week 2 gives you a complete (but incorrectly ordered) example of how to write strings, including the new line added by print. Remember also that Python will print a new line after printing the string.
- In a text file called exercise1.txt, list the following elements that appear in your MIPS code:
    - Line number of every MIPS instruction and the purpose of the instruction.
    - The name of any label.
    - Line number of any assembly directive and the purpose of the directive.

- Name of any global variable.
- Name of any general purpose register used.

Exercise 2

When we discussed the MIPS architecture in the lesson, things might have been a little abstract. In this question, we will use the MIPS simulator to make those concepts more concrete. Run the MIPS code you wrote in the previous exercise as follows:

1. Start by assembling the code you wrote in exercise1.asm file. If there are no errors in your code, you should see that the interface of MARS switches from **Edit** to **Execute**.
2. Once you are in the **Execute** tab, take a moment to look carefully at all the elements in the graphical interface. You will see that some of them correspond to concepts that we discussed in the architecture. In particular, the **Text Segment** and **Data Segment** in the memory; as well as the **Registers**, which are part of the microprocessor. See Figure below.
3. You can now run your program all at once (play button), just to check that it works
4. Run the program step by step. In a text file called exercise2.txt answer the following questions:
    - What is the initial value of Register **PC**, see how **PC** changes as you step through the program. What is the content of **PC** referring to?
    - Look now at the **text segment**. Is there a pattern in the addresses that correspond to each instruction?
    - Has every MIPS instruction been assembled into a single machine code instruction?
    - What is in the **data segment**? Maybe it helps to click the checkbox **ASCII**.
5. Look at the content of the registers and see how it changes when you run the program step by step.



Exercise 3

Now for something a little bit more complex, to practice reading, writing and mathematical operations.

```
integer1 = int(input("Please enter the first integer: "))
integer2 = int(input("Please enter the second integer: "))
quotient = integer1 // integer2
remainder = integer1 % integer2
print("The quotient is " + str(quotient))
print("The remainder is " + str(remainder))
```

Run your MIPS program step by step, paying attention to how the registers change.

Sample output

See sample_output1.txt and sample_output2.txt for sample output.

Important

A translation from high-level code (say in Python) into MIPS is faithful, if it is done by translating each line of code independently of the others (that is, without optimizing the code by reusing the value of registers computed in previous instructions). While this can introduce considerable space/speed costs, it does make the translation easier, thus reducing the chances of committing mistakes.

Hint: You might want to have a look at how to use the LO and HI special registers for this.

Exercise 4

Lets practice by adding more maths. A Pythagorean triple consists of three positive integers, **a**, **b**, and **c**, such that:

$$a^2 + b^2 = c^2$$

Write in file almost_pythagorean.asm a properly commented MIPS program that reads in two positive integers, m and n, and prints the (almost) Pythagorean triple:

$$a = m^2 - n^2, b = 2mn, c = m^2 + n^2$$

We say almost because in a true Pythagorean triple,

$$a = |m^2 - n^2|$$

but we have not used the absolute value for **a**, as you would need to know about branching instructions for doing that (you will learn about this in week 2).

Sample output

See sample_output1.txt and sample_output2.txt for sample output.

Important

If you decide to use global variables to store **a** and **b** (which is not needed, by the way, and I would recommend not to), do not call them that, as MARS is not very good at distinguishing label **b** from the name of the branch pseudo-instruction.