

# 2.2 - Week 2 - Applied - Practical

---

## Introduction



### Objectives

- To be able to write very simple MIPS programs.
- To familiarise yourself with a [MIPS simulator \(MARS\)](#).
- To understand how the MIPS architecture relates to the assembling and executing of simple MIPS programs.

---

## 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 but we recommend 4) 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)`). Note that this will be required for future assessments.

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 - Printing Strings

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

```
print("You're a wizard, Harry")
print("I'm a what?")
```

- 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:
  - The line number of every MIPS instruction and the purpose of the instruction.
  - The name of any label(s).
  - The line number of any assembly directive(s) and the purpose of the directive.
  - Name of any global variable(s).
  - Name of any general-purpose register(s) used.



Manually running your code in the Ed Terminal





You can run your code using the following command:

```
java -jar Mars4_5.jar exercise1.asm
```

## Exercise 2 - Theory

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 copying and pasting the code you wrote in `exercise1.asm` in the last task to MARS and assembling the code you wrote in MARS. 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? Why/Why not?
-  What is actually stored in the **data segment**? HINT - Maybe it helps to click the checkbox **ASCII**.

---

## Exercise 3 - Mathematics

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
product = integer1 * integer2

print("The quotient is " + str(quotient))
print("The remainder is " + str(remainder))
print("The product is " + str(product))
```

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 - Squared Fun

Lets practice by adding more maths. We are going to use the famous high school mathematical equations:

$$\text{sum} : (a + b)^2 = a^2 + b^2 + 2ab$$

$$\text{difference} : (a - b)^2 = a^2 + b^2 - 2ab$$



You should calculate the sum and difference as the **expanded formulas (RHS)**. Solving them using the square of their sum and difference is **not permitted**.

You should then present the output as

```
The sum is <sum> and the difference is <difference>
```

Showing the output of the values of sum and difference. A python file containing the code has been provided to you for reference.



Example

```
Enter the value for a: 4
```

```
Enter the value for b: 6
```

```
The sum is 100 and the difference is 4
```

```
-- program is finished running --
```

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



Important: Please **DO NOT** use the variable names as 'a' and 'b'. You should use variable names that explain the values held in the variables. For example, `a_value`, `b_value`, etc. You can use global variables to store the values of `a^2`, `b^2` and `2ab`. However, you can also just store them in registers.

---

## Advanced Question (optional)

This question is not mandatory. It is for those who have time and are enjoying MIPS. Write a MIPS program in file `exercise5.asm` that reads any two integers and determines if the first one is a multiple of the second one. You are free to choose how to do the output.



Important

For this you will need to look at the Moodle lesson in [Week 2](#), as you will need to implement the equivalent of an if-then-else condition.