Table of contents:

1. Introduction
2. Exercise 1 (string_compare)
    a. Code behavior and description
    b. Pseudocode
    c. Battery Test
    d. Design Decisions
3. Exercise 2 (study_energy)
    a. Code behavior and description
    b. Pseudocode
    c. Battery Test
    d. Design Decisions
4. Exercise 3 (attack)
    a. Code behavior and description
    b. Pseudocode
    c. Battery Test
    d. Design Decisions
5. Exercise 4
6. Problems and conclusion

Authors of the document:

- Eduardo Alarcón: 100472175, group 89 email: 100572175@alumnos.uc3m.es
- Pol Gómez: , 100462234, group 89 email: 100462234@alumnos.uc3m.es

1. Introduction:

To complete this project, we used the online tool known as creator. The architecture used on this project is RISC-V (32 bits). The assignment consists of 4 exercises, 3 involving programming in assembler and the last exercise in an explanation of a theoretical improvement on the last exercise.

2. Exercise 1 (string_comapre):

1. Code behavior and description:

String_compare is a function which compares two given strings, in order to determine whether these two strings are the same or not. If the two strings are the same, the function will return 1, but if the strings are different, string_compare will return 0, and if either of the two strings are empty it will return -1.

When calling string_compare, the function will load in the register x15 the first character of one string, and then it compares the character with the register zero. If the characters are equal to the register zero, it means that the function has reached the end of the first string.If the character is different then, it will return an error, outputing in a3 a -1.

2. Pseudocode:

We created a code that returns in the register a3 a value of 1 if the first character of the string we want to test is the same as the first character of the string stored as password. this function will return a 0 if the characters are different and a -1 or FFFFFFFF if either of the strings is empty.

The process that we follow during this code is:

- Store the address of the first character string tested on a register
- Store the address of the first character string with what we compare on a register
- Store the characters values for both strings
- Do a check for the initial value of the first character of both strings, if either is zero, return -1
- Add their ascii values
- If the result is 0:
    - They have both reached the end of the strings, since the sum of two zeros is zero, and the two strings are equal, return 1
- Else:
- Compare the first character with each other, if they are true, go to the next characters
- If they are different, return 0

3. Battery tests:

The test we ran on our code are the following:

Input data:	**Test description:**	**Expected output:**	**Actual output:**
string1 = “a” string2 = “a”	Comparing two strings with one	1	1

	character only		
string1 = “” string2 = “some”	Comparing string1 being empty	-1	-1
string1 = “some” string2 = “”	Comparing string2 being empty	-1	-1
string1 = “one” string2 = “ons”	Comparing similar string, but different last character	0	0
string1 = “ONE123” string2 = “ons456”	Test including capital letters and numbers	0	0
string1 = “@” string2 = “@”	Comparing special characters, not from a-z	1	1
string1 = “helloñ” string2 = “helloñ”	Comparing special characters while being equal	1	1

4. Design Decisions:

We took the design approach of comparing each character by getting the address of each of the strings and adding one to the directions as we went along, instead of adding an n number of times, for an index.
We decided to store the addresses of string1 on the register a1 and string2 on the register a2, while the contents of the address are stored in a4 and a5. We know that comparing two registers is slower than doing an addition, that is why we chose to do an addition of the two registers to check for the end of the string, instead of doing it one by one.

3. Exercise 2 (study_energy):

1. Code behavior and description:

Study energy is a function that only has as a parameter, the string we are going to compare, treated as the password. The characters that are going to be used are generated by the code, starting with an “a” which in ascii is 97, so we increment by one each time we need the next character.

Study_energy is a function that compares two characters, and counts how many cycles that comparison has taken. Then, it also prints each of the characters that have been tested with the format letter_tested : number_of_cycles (for example b:9). It is based on the string compare function, but some modifications have been made to reduce the number of cycles, for example the assignment of the register a3 with the 0, -1 or 1 depending if the string was different, an error occurred or the string was equal respectively. If the comparison takes longer than another one, we know that is the correct character

As what we are testing is the number of cycles, we can reduce the code even further, by deleting the tags correct, wrong and error and replacing them by the same one. We also store the : as a value ascii (58). Since we no longer require an output of 0,1, or -1.

2. Pseudocode:

The code being analyzed will follows this pseudocode:

- Getting the address of the password
- Create the first letter using ascii
- Start reading the cycles
- String compare
- Stop reading cycles
- Subtract cycles
- Report the letter with the format letter: #cycles {example: d:14}
- Go to next letter

3. Battery tests:

The test we ran on our code are the following:

Input data:	**Test description:**	**Expected output:**	**Actual output:**
password = “”	sintering with no characters, empty string	all characters same number of cycles	all characters same number of cycles
password = “a”	string with one character	a:<n b-z: n	a:14 b-z: 9

password = “z”	string with the last character	z:>n a-y: n	z:14 a-y: 6
password = “hg”	String with two different elements, only the first one should be considered	h:>n a-g U i-z: n	h:14 a-g U i-z: 9
password = “zz”	string with two of the same characters	z:>n a-y: n	z:14 a-y: 9
password = “sdfgh”	String with more than 2 letters	s:>n a-r U t-z: n	s:14 a-r U t-z: 9
password = “ñ”	string with element not in range of the compared elements, ñ does not exists in a-z	a-z : n	a-z:9

4. Design Decisions:

We decided to not store the characters in the data section of the program, as it is more resource friendly, not needing as much memory to execute the program. When doing the report, we must do four different prints, one for the character testing, another for the “:” character, another for the number of cycles, and one final one to change the line for the next character, that is the reason why our report is a bit long.

If, for any reason, the characters that are wanted to be tested change, the only thing that should be done is changing the registers' ascii value. For example, if we wanted to test the capital letters, we would use a1 = 65 and t6 = 90, or if we wanted to test all the characters that can be typed in a password (excluding space), we would use a1=33 and t6=126

4. Exercise 3:

1. Code behavior and description of attack:

The code behind the attack function is very similar to the study energy one. The only sections that change are some optimizations of the code and the reporting of the cycles. Instead of printing them on the screen, we do a small comparison and some conditionals to ensure we get the highest number of cyclers for each character. And lastly, we had to modify the function to make it recursive over a string.

2. Pseudocode:

The code being analyzed will follows this pseudocode:

- Initialize all the registers needed to their corresponding values including letter “a” (the start), letter “z” (the ending)
- Getting the address of the password
- Create the first letter using ascii
- Start reading the cycles
- String compare
- Stop reading cycles
- Subtract cycles
- If there have not been any savings to the memory done yet: (specifically for the first character “a”)
    - Store that letter and cycles into the registers
- Else:
    - Compare them with the stored cycles:
    - If it is greater: (only for character “z”)
        - Store the character value of the current character to the memory
        - Update the pointer of the dummy for the next character to be put in the memory
    - If it is smaller: (only for character “a”)
        - Store the character value of the previous character to the memory
        - Update the pointer of the dummy for the next character to be put in the memory
    - If its equal:
        - Move to the next character to compare with the next cycles
- Repeat until a character is found with different energy cycles
- Go to the next character of the password we are guessing and repeat the process from step 1

3. Battery tests:

The test we ran on our code are the following:

Input data:	Test description:	Expected output:	Actual output:
password = “abcd”	string with less than 8 characters	not contemplated in exercises	abcd
password = “a”	string with only one character	not contemplated in exercises	a
password = “aaaabbbb”	string with 8 characters	aaaabbbb	aaaabbbb
password = “mmmmmmmm”	string with 8 equal characters	mmmmmmmm	mmmmmmmm
password = “”	empty string	nothing, as the dummy is already filled with 0s	nothing, as the dummy is already filled with 0s
password = “abcgeccf”	String with equal consecutive and going from a smaller to larger ascii and vice versa	abcgeccf	abcgeccf
password = “ñ”	string with a character that is not in range a-z	not contemplated in exercises	stops because no character was found

4. Design Decisions:

The only problem we encountered when doing this exercise was the comparison of the string as well as storing the values into the memory. The choosing of the highest cycle count is a bit confusing: If no value has been saved previously, that value is saved, in other words, the cycle count of “a” is always stored and then it is compared with the rest of the characters. If it is higher than the next one, the stored one, the “a” is pushed to memory. If it isn’t, then it skips till a character has a higher number of cycles. We decided to not limit the range of the password to 8 characters, as this program could work with any password of length n as long as there is sufficient memory to store it.

Additionally this program can be modified to fit ranges of characters other than “a-z”. For example, if we wanted to test the capital letters, we would use a3, = 65 and t6 = 90, or if we wanted to test all the characters that can be typed in a password (excluding space), we would use a3, =33 and t6=126, however since the exercise asked for characters only from “a-z” the values are a3= 97 and t6 = 122, fitting the “a” and “z” ascii values respectively.

5. Exercise 4:

Firstly, in order to nullify an energy cycle attack, the function string_compare could be modified by adding dummy cycles to the comparing of an incorrect character to make the number of cycles even. This would in turn allow for the answer of the energy cycle attack to not function, since all the characters would show the exact same amount of energy cycles, this would not let the attack determine any specific character to do more energy cycles than another, thus breaking the attack.

Another option we have, as a general idea, is better to store the passwords hashed, so even if the characters can be known, it is almost impossible to know the original password. Hashing a password allows you to have that password encrypted in a non-reversible way (it can be reversed except it would take too much time for it to be viable), and with a specific amount of characters- While the attack is technically still possible, the result would give the hashed password, and that is essentially useless since it can not be reversed.

6. Problems and conclusion:

Generally, we did not find any complication beyond what could be expected, and we produced solutions for all the examples. We struggled to think of the most efficient way to code the programs, however with time and dedication we came up with the most efficient solutions (in our opinion).

We encountered problems with storing data into the memory as well as going step by step into the code without really understanding some of the functions before reading the documentation. All in all, we mostly encountered problems with exercise 3, as the functions were specifically made and needed some tweaking, and we had to juggle with the values carefully and efficiently which proved to be more difficult than expected.